DUAL EULERIAN GRAPHS WITH APPLICATIONS TO VLSI DESIGN

by

Andre Freeman

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Applied Mathematics

April 2003

APPROVED:

Dr. Brigitte Servatius, Thesis Advisor

Dr. Bogdan Vernescu, Head of Department

Abstract

A Dual-Eulerian graph is a plane multigraph G that contains an edge list which is simultaneously an Euler tour in G and an Euler tour in the dual of G. Dual-Eulerian tours play an important role in optimizing CMOS layouts of Boolean functions. When circuits are represented by undirected multigraphs the layout area of the circuit can be optimized through finding the minimum number of disjoint dual trails that cover the graph. This paper presents an implementation of a polynomial time algorithm for determining whether or not a plane multigraph is Dual-Eulerian and for finding the Dual-Eulerian trail if it exists.

Acknowledgements

I would first off like to thank my thesis advisor, Professor Brigitte Servatius, for her support and encouragement. I also thank my family, for believing in me and for keeping me focused and moving forward.

Contents

List of Figures

Chapter 1

Introduction

Identifying dual trails and tours in plane multigraphs is of particular interest to designers of CMOS VLSI circuits. These circuits are represented as undirected multigraphs, and a particular problem in VLSI design is optimizing the layout using only the proprties of the given multigraph. An optimal layout exists when one trail simultaneously covers both the multigraph G and its dual G^*, which occurs when a multigraph is Dual-Eulerian. Creating efficient algorithms for determining whether or not a multigraph is Dual-Eulerian and identifying Dual-Eulerian trails is therefore very important.

A multigraph G contains an Euler tour if there exists a tour in G that contains every edge of G. A multigraph is Eulerian if it contains an Euler tour or an Euler trail. Suppose G is a plane connected multigraph and G^* is the geometric dual of G. If the same sequence of edges form an Euler tour or an Euler trail in G and simultaneously in G^*, G is called Dual-Eulerian.

It is important to note that both G and G^* may be Eulerian but this does not guarantee that they are Dual-Eulerian. As an example consider the plane graph G and its dual G^* in Figure 1.1 where G is denoted by straight lines and G^* is represented by dotted lines. G contains the Euler tour $\{a, b, c, d, e, f, g, h, i, a\}$ and G^* contains the Euler tour $\{a, i, f, e, d, g, h, c, b\}$ but these are not the same sequences of edges, and no such sequence can be found. Therefore it is not sufficient to simply search for Eulerian trails in either G or G^* in order to determine whether or not G is Dual-Eulerian.

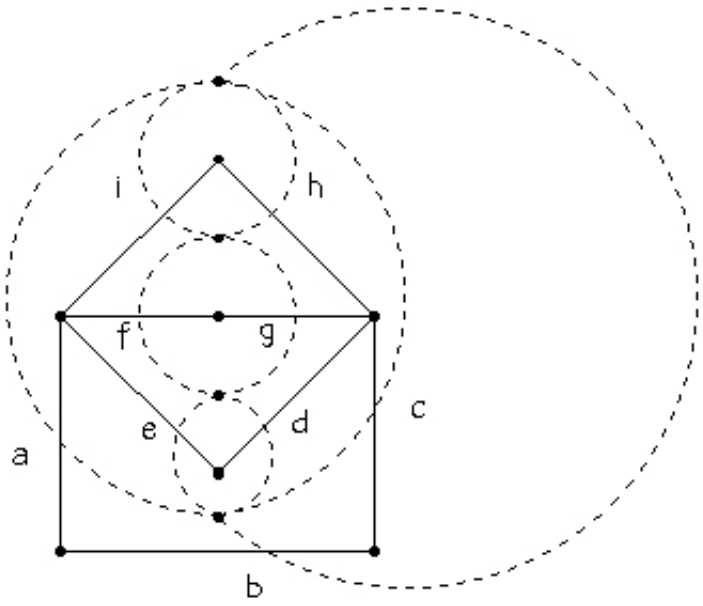

Figure 1.1: Multigraph G and its dual G*

Determining whether or not a multigraph admits a Dual-Eulerian embedding is non-trivial[1,8,9]. A

multigraph may have several plane embeddings and each plane embedding has its own corresponding geometric dual. One embedding may be Dual-Eulerian while another may not. Figure 1.2 demonstrates this idea by showing two different planar representations of the same multigraph in which the planar representation in (a) is Dual-Eulerian while the planar representation in (b) is not because no Euler-Petrie trail exists. Figure 1.3 displays two different planar representations of the same graph which are both Dual-Eulerian. A simpler problem to tackle is whether or not a plane multigraph admits a Dual-Eulerian trail and several algorithms have been presented to solve this problem. I employ an algorithm introduced in [8] and [9] which searches for Euler-Petrie trails to determine whether a plane multigraph is Dual-Eulerian. A key property of Petrie trails is that the same sequence of edges in G is also a sequence of edges in G^*. Finding an Euler-Petrie trail, or tour, corresponds to finding a Dual-Eulerian trail, or tour. Therefore if a Euler-Petrie trail or tour exists, then the plane multigraph is Dual-Eulerian, and if no such Euler-Petrie trail exists then the plane multigraph is not Dual-Eulerian.

This paper presents an implementation of a polynomial time algorithm for determining whether or not a plane multigraph is Dual-Eulerian, and for finding the Dual-Eulerian trail if it exists. The al-

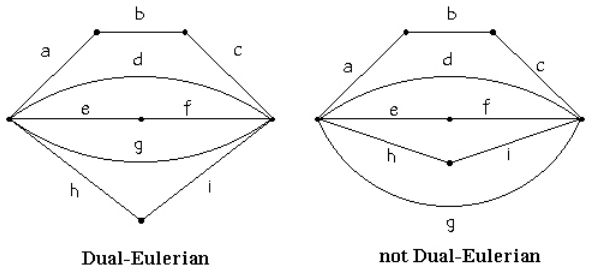

Figure 1.2: Two different planar representations of the same multigraph G

gorithm accepts as input the adjacency structure of a multigraph G and proceeds to partition the multigraph into biconnected components and then into triconnected components forming the *3-BlockTree* of G. The algorithm constructs a planar representation of each triconnected component and then merges the separate planar representations into one plane multigraph. Once this is accomplished the final step is to check the specific planar representation for Euler-Petrie trails using the Euler-Petrie algorithm presented in [9].

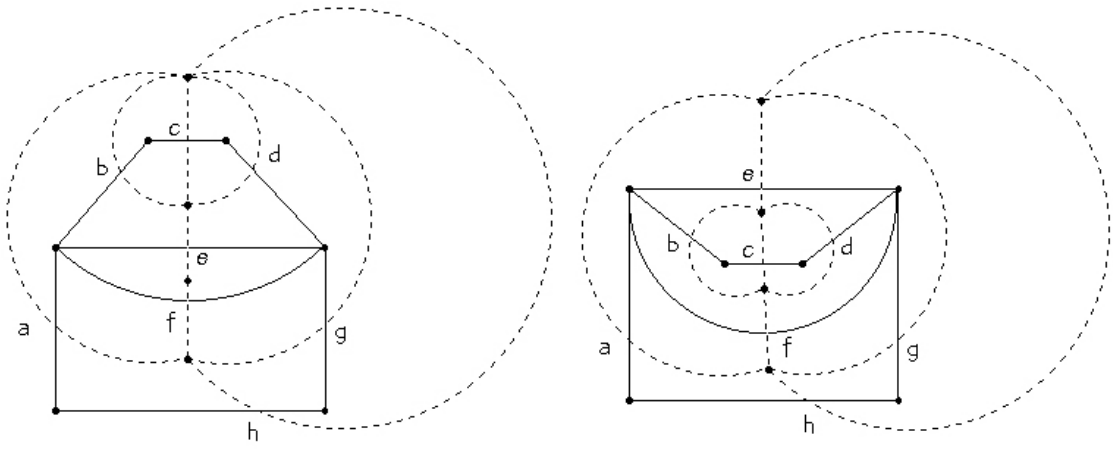

Figure 1.3: Two different Dual-Eulerian planar representations of the same multigraph G

Chapter 2

Background

Before continuing to discuss the algorithm in detail, let's first discuss some basic terminology which will be used throughout the remainder of this paper. The following definitons are taken from [3,7,11].

A *graph* G consists of a set V of *vertices* and a set E of *edges*, written $G = (V, E)$. A *directed edge*, also called an *arc*, corresponds to an ordered pair of distinct vertices (u, v), where u is the *tail* of the edge and v is the *head* of the edge. A graph is *directed* if every edge is directed. Directed graphs are also known as *digraphs*. A *multiedge* is a set of at least two edges, all of which have the same endpoints. A *multigraph* is a graph with multiedges.

If (v, w) is an edge of a graph G then vertices u and v are said to be *adjacent*. An edge (v, w) is *incident* to vertices v and w, and v and w are *incident* to (v,w). A *walk* is an alternating sequence $\mathbf{v}_0$,$\mathbf{e}_1$,$\mathbf{v}_1$,$\mathbf{e}_2$,...,$\mathbf{e}_j$,$\mathbf{v}_k$ of vertices and edges where consecutive edges are adjacent, so that each edge $\mathbf{e}_i$ joins vertices $\mathbf{v}_{i-1}$ and $\mathbf{v}_i$. A *directed walk* is an alter-

nating sequence of vertices and arcs $\mathbf{v}_0$,$\mathbf{e}_1$,$\mathbf{v}_1$,$\mathbf{e}_2$,...,$\mathbf{e}_j$,$\mathbf{v}_k$ where the arcs align head to tail, so that each vertex is the head of the preceding arc and the tail of the subsequent arc.

A *trail* is a walk in which no edge occurs more than once. A *directed trail* is a directed walk in which no arc is repeated. A *path* $p : v \to w$ is a trail in which all of its vertices are different, except that the initial and final vertices may be the same. A *directed path* is a directed trail in which no vertex is repeated. A *closed walk (trail or path)* is a walk, trail, or path whose starting and ending vertex are the same. The *length* of a walk is the number of edges in the walk. A *cycle* is a closed walk of positive length. Two cycles which are cyclic permutations of each other are considered the same cycle.

A graph is *connected* if it has the property that each pair of edges is connected by a path. Given a graph G, a *subgraph* of G is a graph H whose vertices and edges are all in G. If $G = (V, E)$ and $G\prime = (V\prime, E\prime)$, and $G\prime \subset G$ and $V\prime \subset V$, then $G\prime$ is a subgraph of G. A *tree* is a connected graph without any cycles as subgraphs. A *spanning tree* T of a multigraph G is a tree whose vertex set contains all the vertices of G. A graph is *planar* if the graph has the property that it can be drawn in the plane without any edge crossings.

An *Euler trail* is trail that contains all the edges of a graph. An *Euler tour* is a closed Euler trail. An *Eulerian graph* is a graph that contains an Euler tour. An *embedding* of a graph in a surface is a drawing of the graph onto some surface so that there are no edge-crossings. A *region* R of a graph imbedded in a surface is a maximal expanse of surface containing no vertex and no part of any edge of the graph. The *boundary of a region R* of an imbedded graph is the subgraph containing all vertices and edges incident on R.

Given an embedding of a graph in a surface, a *face* is a region plus its boundary. A *dual graph embedding* G^d, is a new graph embedding obtained by placing a dual vertex in the interior of each existing ("primal") region and by drawing a dual edge through each existing ("primal") edge connecting the dual vertices on its opposite sides. The edge sets of G and G^d are identical, and the vertices of G^d correspond to the faces of G and vice versa. A trail in G is a *dual trail* if there exists a trail in the dual graph G^d with an identical edge sequence. Both the trail in G and the trail in G^d are dual trails. A dual trail is a *dual tour* if it is a tour in both G and G^d. G is *Dual-Eulerian* if and only if it contains a dual trail that contains every edge of G.

The *degree of a vertex v*, *deg(v)*, is the number of proper edges (edges with two distinct endpoints)

incident on v. A *disconnecting set of edges* in a connected graph is a set of edges whose removal yields a disconnected graph. A *bond* is a minimal disconnecting set of edges.

Chapter 3

Depth First Search

There are several ways of storing vertex and edge information into a computer. The most common method is to represent a graph by an adjacency structure in the form of *adjacency lists* or an *adjacency matrix*. An adjacency list, or adjacency set, $A[v]$ of a vertex $v \in G$ lists all vertices in G which are adjacent to v. An *adjacency structure* A of a graph G is the set of adjacency lists that corresponds to all the vertices $v \in G$. An adjacency matrix A_{ij} of G can be defined as follows. If an edge (i, j) exists in G, then $A_{ij} = A_{ji} = 1$, and if no edge exists between vertices i and j then $A_{ij} = A_{ji} = 0$. Both adjacency lists and matrices can also represent multiedges. One way in which adjacency lists can store a multiedge (i, j) with n edges is by placing n occurrences j in $A[i]$ and similarly placing n occurrences of i in $A[j]$. An adjacency matrix can store a multiedge (i, j) with n edges by setting $A_{ij} = A_{ji} = n$. In certain situations all that may be known about a given multigraph is the multigraph's adjacency structure.

In order to determine the connectivity of a multigraph, or create a visual representation, efficient procedures are needed to extract information about the multigraph from it's adjacency structure. One such method which is used extensively to extract information from a graph solely from the graph's adjacency structure is a *depth-first search*.

The depth-first search presented in this paper was introduced by Tarjan in [5,10]. A depth-first search is a recursive procedure that traverses all the edges of a connected graph G using the information stored in all the adjacency lists $A[v]$. This procedure creates an ordering ($NUMBER[v]$) on all $v \in V$ based on when each vertex was initially traversed during the search. A depth-first search is executed as follows. Initially, all the vertices v are unexplored and have $NUMBER[v] = 0$. Starting from any vertex v, label v explored by setting $NUMBER[v] = 1$, specifying that v is the first vertex explored. Next find a vertex $w \in A[v]$ that has not been explored. Label w explored by numbering it according to the order which it was explored. The process continues until all the vertices are numbered. Each time a vertex $w \in A[v]$ is identified, the edge (v,w) is traversed. As this occurs, the procedure checks whether or not $NUMBER[v] < NUMBER[w]$, i.e. whether or not v was explored before w in the search. If $NUMBER[v] < NUMBER[w]$, the edge (v, w) is labeled a *tree edge*

and is denoted $v \rightarrow w$. If $NUMBER[v] > NUMBER[w]$, the edge (v, w) is labeled a *frond* and is denoted $v- \rightarrow w$. The procedure also performs a check to ensure that the an edge is not traversed and added in both directions. A depth-first search thus partitions the edges of G into two sets, a set of tree edges T and a set of fronds F. A *palm tree* P is a partition of the edges of G into two sets of directed edges: tree edges T and fronds F. The set of tree edges forms a spanning tree T of G, where each edge (u,v) has $NUMBER[u] < NUMBER[v]$. The set of fronds satisfy the opposite property such that for each edge (u,v) has $NUMBER[u] > NUMBER[v]$.

To illustrate a depth-first search, consider the following graph G in Figure 3.1.

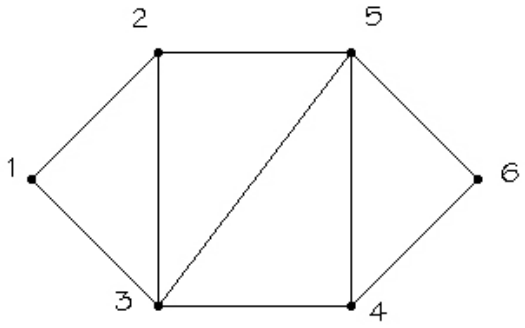

Figure 3.1: G=(V,E), |V| = 6, |E| = 9

The adjacency structure of G is shown below where the adjacency lists are ordered in increasing order.

$$A[1] = \{2, 3\}, A[2] = \{1, 3, 5\}, A[3] = \{1, 2, 4, 5\}$$
$$A[4] = \{3, 5, 6\}, A[5] = \{2, 3, 4, 6\}, A[6] = \{4, 5\}$$

A depth-first search beginning at vertex 1 will proceed as follows. Initially all the vertices are unexplored, thus $NUMBER[1...6] = 0$**. Vertex 1 will be numbered first,** $(NUMBER[1] = 1)$**. The next vertex to be explored will be the first vertex in the adjacency list of vertex 1 which has not been explored, namely vertex 2. Therefore** $NUMBER(2) = 2$ **and since** $NUMBER[1] < NUMBER[2]$ **the edge** $(1, 2)$ **is labeled a tree edge. The first vertex in the adjacency list of vertex 2 is vertex 1. Since vertex 1 has already been explored, the algorithm checks whether or not the edge** $(1, 2)$ **has been traversed. Since** $(1, 2)$ **was previously traversed, the procedure skips to the next vertex in** $A[2]$**, namely vertex 3. Since** $NUMBER[3] = 0$**, vertex 3 is explored (**$NUMBER[3] = 3$**) and the edge** $(2, 3)$ **is added. Since** $NUMBER[2] < NUMBER[3]$ **the edge** $(2, 3)$ **is labeled a tree edge. The first vertex in the adjacency list of vertex 3 is vertex 1.** $NUMBER[1] < NUMBER[3]$**, and the edge** $(1, 3)$ **has not been traversed. Therefore the edge** $(3, 1)$ **is added and labeled a frond. The procedure then finds the next unexplored vertex adjacent to vertex 3, namely vertex 4. Vertex 4 has yet to be explored, thus (**$NUMBER[4] = 4$**), and the edge** $(3, 4)$ **is added. Following is a list of the edges in the order they are explored.** $(1, 2), (2, 3), (3, 1), (3, 4)$

The palm tree generated by a depth-first search of G is shown in Figure 3.2. The tree edges T are presented by bold lines, and the fronds F are presented by dashed lines.

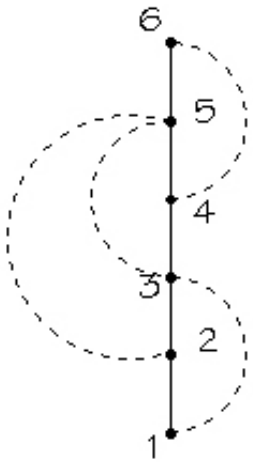

Figure 3.2: Palm tree P of G

The following notation is used throughout the next two sections. If (u, v) is a tree edge in P, then we denote this relation by $v \rightarrow w$. If $v \rightarrow w$ then v is the *father* of w, and w is the son of v. The relation, "there is a path from v to w in T", is denoted by $v \rightarrow *w$. If $v \rightarrow *w$ then v is an *ancestor* of w, and w is a *descendant* of v. If v is a vertex in a tree T, T_v is the subtree of T containing all the vertices that are descendants of v in T. If a path p exists between vertices u and v in G, then this is denoted by $p : v \Longrightarrow w$.

The palm tree P generates a visual representation

of G and there are algorithms for determining the planarity of a multigraph using P. The planarity algorithm presented in [5] separates a multigraph into biconnected components, generates a palm tree for each biconnected component, and then examines each biconnected component individually for planarity. The algorithm identifies a cycle in the palm tree (of the biconnected component), which consists of a sequence of tree arcs followed by one frond, and embeds it. The algorithm then deletes the cycle producing a set of disconnected segments. The algorithm then recursively embeds each segment checking at each step that the new embedding created by adding the disconnected segments is planar. The number of steps required for this planarity algorithm is linear with respect to the number of vertices and edges in the multigraph.

Chapter 4

Biconnected Components Algorithm

A connected multigraph G is *biconnected* if for each triple of distinct vertices $v, w, a \in V$, there exists a path $p : v \implies w$ such that a is not on the path p. If there is a distinct triple v, w and a in G such that a is on every path from $p : v \implies w$, then a is called an *articulation point* or *separation point* of G and G is not biconnected. Figure 4.1 displays a non-biconnected graph with an articulation point at vertex 3. From the graph we can see that every path $p : 1 \implies 4, p : 1 \implies 5, p : 2 \implies 4$ and $p : 2 \implies 5$ must pass through 3.

The vertex c thus partitions the graph into two biconnected graphs. One component is the graph consisting of the edges $\{(1,2)(1,3)(2,3)\}$ and the other biconnected component consists of the edges $\{(3,4)(3,5)(4,5)\}$. Tarjan presents a linear time algorithm for finding the biconnected components of an undirected graph by identifying the articulation points [10]. The articulation points and corresponding biconnected components are found through applying a

depth-first search on G and creating a palm tree. In addition to ordering the vertices of G, the procedure creates an array $LOWPT[v]$, which stores the lowest vertex reachable from every vertex after traversing 0 or more tree arcs followed by one frond.

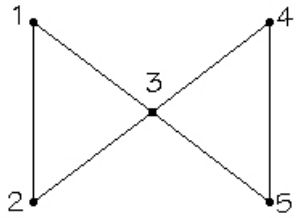

Figure 4.1: $G = (V, E)$, |V|=5, |E|=6

The following condition is used to find the articulation points. Let G be an undirected graph. Suppose P is a palm tree and T is the spanning tree contained in P. Suppose a, v, w are distinct vertices in G such that $(a, v) \in T$, w is not a descendent of v, and $LOWPT[v] \geq a$. Then a is an articulation point in G, and the removal of a will disconnect G.[10]

It is clear to see that this condition holds. If $(a, v) \in T$, then a was traversed before v in the depth-first search, so $NUMBER[a] < NUMBER[v]$. If $LOWPT[v] \geq a$, then every path (sequence of tree arcs followed by one frond) starting from v will either remain in T_v. Since w is not a descendent of v, w cannot be in T_v. Any path $p : w \implies v$ must pass through a, therefore a is an articulation point. The biconnected component corresponding to the

articulation point a is the subtree T_v.

The biconnected component algorithm stores the edges in a stack as they are traversed, and when an articulation point is found the edges on the stack corresponding to the biconnected component are removed. This set of edges forms a biconnected component. The biconnectivity algorithm is efficient as it requires only one depth-first search procedure to identify all of the articulation points and the corresponding biconnected components.

To illustrate this algorithm, consider the graph G in Figure 4.2. The graph G is displayed with its palm tree P as well as with its biconnected components.

As the palm tree P is constructed, the articulation points and biconnected components are found simultaneously. The edges are partitioned into the following two sets.

$$T \ni$$
$$(1,2)(2,3)(3,4)(4,5)(5,6)(6,7)(7,8)(8,9)(9,10)(10,11)(11,13)(13,12).$$
$$F \ni (3,1)(4,2)(6,4)(7,5)(8,6)(9,7)(11,9)(12,10).$$

The algorithm produces the following LOWPT[v] values:

$$LOWPT[1] = 1, LOWPT[2] = 1, LOWPT[3] = 1, LOWPT[4] = 2,$$
$$LOWPT[5] = 4, LOWPT[6] = 4, LOWPT[7] = 5,$$
$$LOWPT[8] = 6, LOWPT[9] = 7, LOWPT[10] = 9$$
$$LOWPT[11] = 9, LOWPT[12] = 10, LOWPT[13] = 10$$

Beginning with the starting vertex, (vertex 1), the procedure identifies vertices adjacent to vertex 1, and if these adjacent vertices are unexplored this new edge becomes a tree edge. The algorithm then continues with the adjacent vertex, and finds new vertices adjacent to it. At some point however, the procedure will return to vertex 1 and search for other neighboring vertices. The act of returning to a vertex after a tree edge has been started from it, is called backing up along an edge. When the edge $(9, 10)$ **is backed up along, the algorithm detects that** $LOWPT[10] \geq 9$**. This implies that any path starting from a vertex in** $\mathbf{T}_{10}$ **must pass through vertex 9 in order to reach the *proper ancestors* of 9 (ancestors of 9≠9). Since** $NUMBER[9] = 9$**, (9 is the ninth vertex explored), the algorithm knows that there exists vertices that are not descendants of 9 (namely vertices numbered 1,2,3,4,5,6,7 and 8). Thus the edges in** $\mathbf{T}_{10}$**, specifically** $(12, 10)$**,** $(13, 12)$**,** $(11, 13)$**,** $(11, 9)$**,** $(10, 11)$**, and** $(9, 10)$ **are all in the same biconnected component and 9 is the corresponding articulation point. Similarly, when the edge** $(4, 5)$ **is backed up along the algorithm detects that** $LOWPT[5] \geq 4$**. Vertex 4 has proper ancestors, there-**

fore every path between vertices in T_5 and the ancestors of 4 must pass through 4. Thus 4 is an articulation point and the edges $(9,7)$, $(8,9)$, $(8,6)$, $(7,8)$, $(7,5)$, $(6,7)$, $(6,4)$, $(5,6)$ and $(4,5)$ belong to a separate biconnected component. The edges that remain, $(4,2)$, $(3,4)$, $(3,1)$, $(2,3)$, and $(1,2)$ belong to the final biconnected component.

The biconnectivity algorithm finds the biconnected components of a graph in time linear with respect to the number of edges and vertices in the graph [10]. The biconnected components are edge-disjoint and the original multigraph can be restored from the biconnected components by the process of vertex identification.

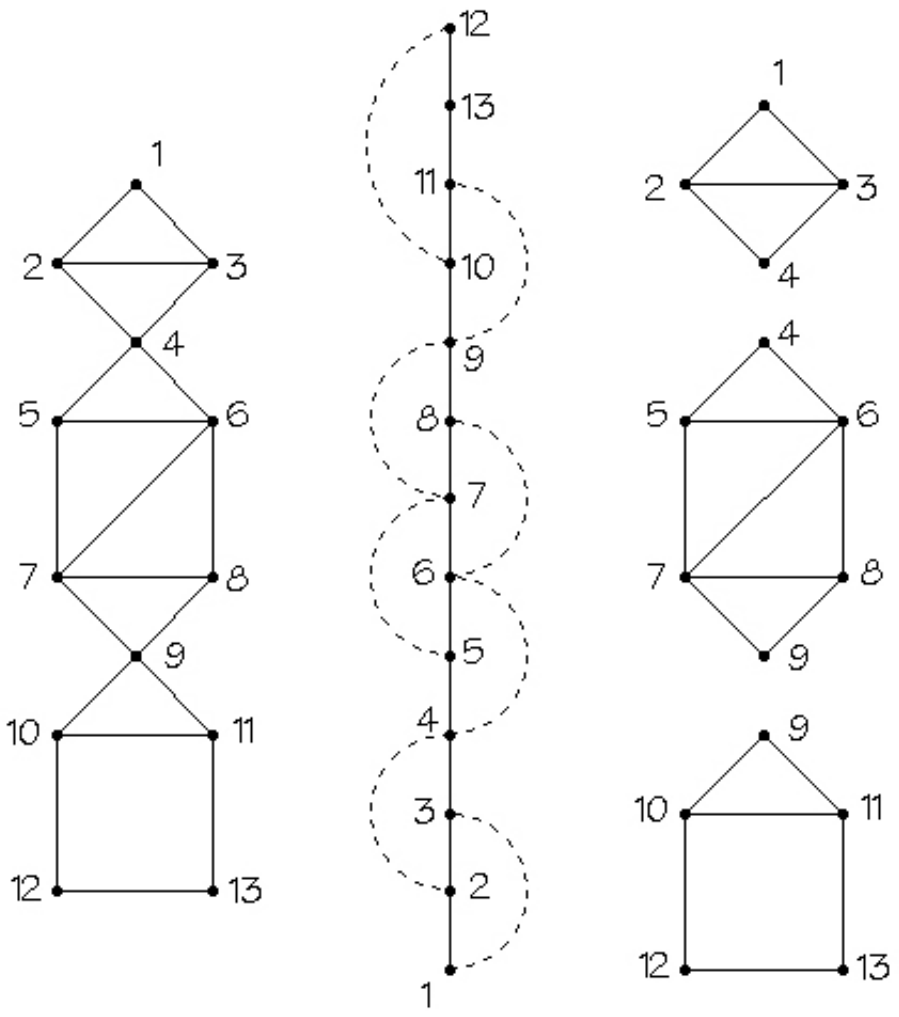

Figure 4.2: Graph G, Palm tree P, and Biconnected Components of G

Chapter 5

Triconnected Components Algorithm

Let $\{a, b\}$ be a pair of vertices in an undirected multigraph G. Suppose E can be partitioned into equivalence classes $E_1, E_2, ..., E\_n$, such that two edges are in the same equivalence class if both edges lie on a common path not containing any vertex of $\{a, b\}$ except as endpoints. If such a pair of vertices $\{a, b\}$ exists, then the equivalence classes $E_1, E_2, ..., E\_n$ are called the *separation classes of G with respect to* $\{a, b\}$. If there exist at least two separation classes with respect to $\{a, b\}$ then $\{a, b\}$ is a *separation pair* of G, unless (i) there are exactly two separation classes and one consists of a single edge, or (ii) there are exactly three separation classes, each consisting of a single edge. Consider the graph in Figure 5.1.The vertex pair $\{1, 2\}$ partitions E into three equivalence classes, namely $\{a, b, i\}$, $\{h\}$, and $\{c, d, e, f, g\}$. Any path containing at least one edge in the set $\{a, b, i\}$, and one of the edges in the set $\{c, d, e, f, g\}$, must contain either vertex 1 or vertex 2. Similarly, any path containing edge $\{h\}$ and any edge

in $\{c, d, e, f, g\}$ or $\{a, b, i\}$, must also contain either vertex 1 or vertex 2. The vertex pair $\{3, 4\}$ is another separation pair which partitions E into two classes, namely $\{a, b, c, g, h, i\}$ and $\{d, e, f\}$. Note that a separation pair $\{a, b\}$ may or may not be an edge in E.

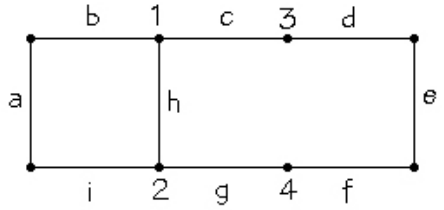

Figure 5.1: Separation pairs 1,2 and 3,4

If G is a biconnected multigraph such that no pair of vertices $\{a, b\}$ is a separation pair, then G is *triconnected*. Suppose $\{a, b\}$ is a separation pair of G. Let E_1 and E_2 be the separation classes with respect to $\{a, b\}$ such that $|E_1| \geq 2$ and $|E_2| \geq 2$. Let $G_1 = (V(E_1), E_1 \cup \{(a, b)\})$ and $G_2 = (V(E_2), E_2 \cup \{(a, b)\})$. The graphs G_1 and G_2 are called the *split graphs of G with respect to* $\{a, b\}$. Replacing a multigraph by two split graphs is called *splitting* G. The new edges (a, b) added to G_1 and G_2 are called *virtual edges*. Each virtual edge gets labeled according to the split operation associated with it.

If G is biconnected, any split graph of G is also biconnected. Suppose a multigraph is split, and the split graphs are split, and this is repeated until no

more splits are possible. The resulting multigraphs are then *triconnected* and are called the *split components of* G. The split components of a graph are not necessarily unique.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two split components both containing a virtual edge (a, b, i), where (a, b, i) was added to G_1 and G_2 during the ith split. Let $G = (V_1 \cup V_2, (E_1 - \{(a, b, i)\}) \cup (E_2 - \{(a, b, i)\})))$. G is called the *merge graph* of G_1 and G_2. Merging is the inverse operation of splitting a graph. Performing a sufficient number of merge operations on the split components of a multigraph recreates the original multigraph. There are three types of split components in a multi-graph: *triangles* $\{(a, b), (a, c), (b, c)\}$, *triple bonds* $\{(a, b\}, (a, b), (a, b)\}$, and *triconnected graphs*. Examples of possible split components are shown in Figure 5.2. The *triconnected components* of a multigraph are formed by merging adjacent triangles and combining triple bonds which share the same pair of vertices. The triconnected components of a multi-graph G are unique[4].

This paper implements an algorithm for finding the triconnected components of a multigraph introduced by Tarjan and Hopcraft[4]. The algorithm accepts as input the adjacency structure of a connected multigraph and is divided into the following steps.

1. Split off multiple edges of G to form a set of

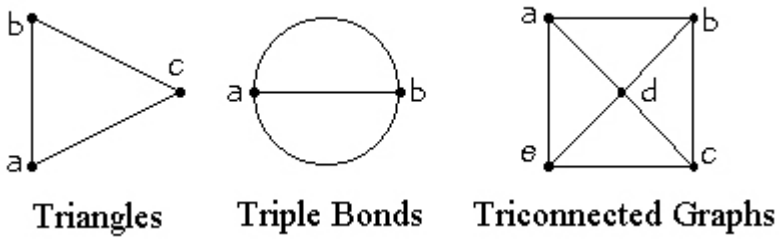

Figure 5.2: Types of split components

triple bonds and a graph G'.

2. Find the biconnected components C of G'.

3. For each biconnected component C of G' :

(a) Find the split components of C.

(b) Combine triple bonds with the same pair of vertices into bonds.

(c) Combine adjacent triangles into polygons.

The previous steps outlines the procedure for finding triconnected components of a graph. The set of triconnected components forms the vertices of the *3-Block Tree* of G. Step 1 in the triconnectivity procedure is accomplished through identifying the multiple edges and immediately splitting them off. The most difficult part of the algorithm is step 3. The biconnected components are found through employing the biconnected components algorithm previously outlined. Once the biconnected compo-

nents are found, the next step is to find the split components of each biconnected component.

For the remainder of this section we will assume that G is biconnected and contains no multiple edges. Similar to the biconnectivity algorithm, the triconnectivity algorithm uses a depth-first search $DFS()$ technique to create a palm tree P of G. For each vertex $v \in V$, the $DFS()$ procedure identifies the values $FATHER[v]$, $ND[v]$, $LOWPT1[v]$ and $LOWPT2[v]$. $FATHER(v)$ stores the predecessor to v in the spanning tree T by keeping track of the vertex leading to it. The only exception is the first vertex (vertex numbered 1) of the search whose father is designated to be itself. $ND(v)$ stores the number of descendents of v in the spanning tree T. $LOWPT1(v)$ stores the lowest vertex reachable from v after traversing one or more tree arcs followed by one frond. $LOWPT2(v)$ stores the second lowest vertex reachable from v after traversing one or more tree arcs followed by one frond. These values will be used first to reorder each vertex's adjacency list to create a set of paths which cover the graph.

Once the $DFS()$ procedure is performed, we use the information gathered to reorder the adjacency list of each vertex. To do this we define a mapping ϕ from each edge in P to a number $x \in \{1, 2, ..., |V|+1\}$. Initially each new adjacency list is empty. Next we add edges to each adjacency list based on the $\phi(u, v)$.

The new adjacency lists are reordered so that the edges appear in increasing value of $\phi(u, v)$. In the new structure, the vertices are ordered such that the first vertex in $A_1[v]$ is the vertex adjacent to v with the smallest $LOWPT2[]$ value, i.e. the vertex adjacent to v which will travel to the lowest numbered vertex after traversing a number of tree edges followed by one frond. If a vertex v has two fronds starting from v, then the frond which reaches the lower numbered vertex will be placed at the beginning of the list, and the frond which reaches the second lowest vertex will be placed next in the list.

Using this new adjacency structure, the $PATHFINDER()$ procedure generates a set of disjoint paths $P_0, P_1, P_2, ..., P_n$ that cover the entire graph. The $PATHFINDER()$ is based on the depth-first search technique but the main difference from this procedure and $DFS()$ is that the former uses the reordered adjacency lists. Each path terminates at the lowest possible vertex reachable from the vertices on the path. The initial path P_0 is a cycle, and every other path is simple and has only its initial and terminal vertex in common with previously generated paths.

Once we have our set of disjoint paths that cover the graph, the next step is to search for separation pairs using the $PATHSEARCH()$ procedure. Each path is traversed in order, and the procedure identifies separation pairs as they are encountered. When

a separation pair is found, the split component associated with that separation pair is removed from the graph. Separation pairs have several key properties. First, if $\{a, b\}$ is a separation pair in G with $a < b$, then $a \rightarrow *b$ in T (i.e. there exists a path in T from a to b). Separation pairs are separated into three types by the following three conditions.

1. If there are distinct vertices $r \neq a, b$ and $s \neq a, b$ such that $b \rightarrow r$, $LOWPT1(r) = a$, $LOWPT2(r) >= b$, and s is not a descendent of r.

(The pair $\{a, b\}$ is called a type 1 separation pair).

2. If there is a vertex r such that $a \rightarrow r \rightarrow *b$; b is a first descendent of r; $a \neq 1$; every frond $x \rightarrow *y$ with $r \leq b$ has $a \leq y$; and every frond $x \rightarrow *y$ with $a \leq y \leq b$ and $b \rightarrow w \rightarrow *x$ has $LOWPT1(w) \geq a$.

(The pair $\{a, b\}$ is called a type 2 separation pair).

3. If (a, b) is a multiple edge of G and G contains at least 4 edges.

Consider the following graph G and its palm tree P in Figure 5.3 and Figure 5.4. The $LOWPT1()$ and $LOWPT2()$ values are listed in the brackets to the right of each vertex in P.

The separation pairs $(2, 3)$ and $(4, 5)$ of the graph in Figure 5.3 are both type 1. $(2, 3)$ is a type 1 separation pair because $(a, b) = (2, 3)$ and since $3 \rightarrow 4 (r = 4)$, $LOWPT1(4) = 2$, and $LOWPT2(4) = 4$. $(4, 5)$ is a

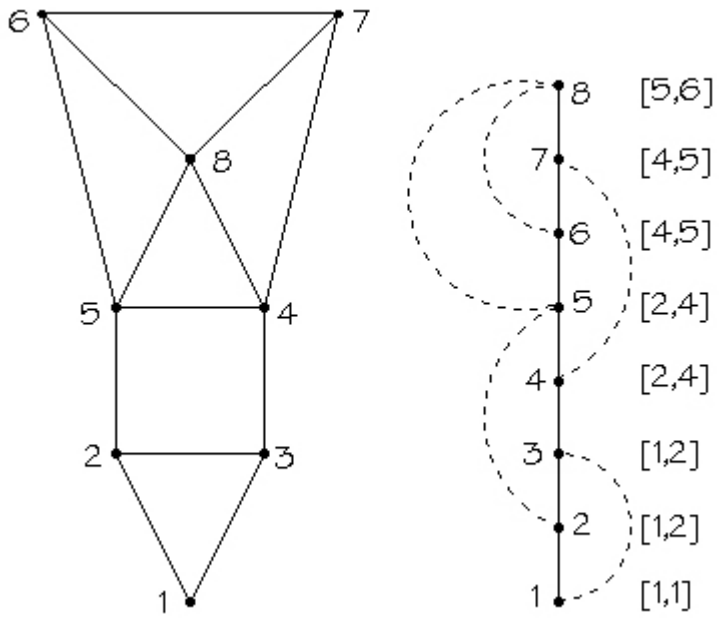

Figure 5.3: Type 1 separation pairs 2,3 and 4,5

type 1 separation pair because $(a, b) = (4, 5)$ **and since** $5 \rightarrow 6(r = 6)$**,** $LOWPT1(6) = 4$**, and** $LOWPT2(6) = 5$**.**

In the graph in Figure 5.4, the separation pairs are $(2, 3)$ **and** $(4, 7)$**.** $(4, 7)$ **is a type 2 separation pair because** $(a, b) = (4, 7)$**, and every frond with** $4 < y < 7$ **and** $7 \rightarrow 8$ **has** $LOWPT1(8) \geq 4$**.The** $PATHSEARCH()$ **procedure is a recursive procedure based on the depth-first search technique.**

From each vertex u**, the procedure identifies vertices adjacent to** u**. Suppose vertex** v **is adjacent to** u**. The procedure determines whether the edge** (u, v) **is a tree edge or a frond. If** (u, v) **is a tree edge, the procedure proceeds to search for vertices adjacent to** v**. Eventually the procedure will return to** u

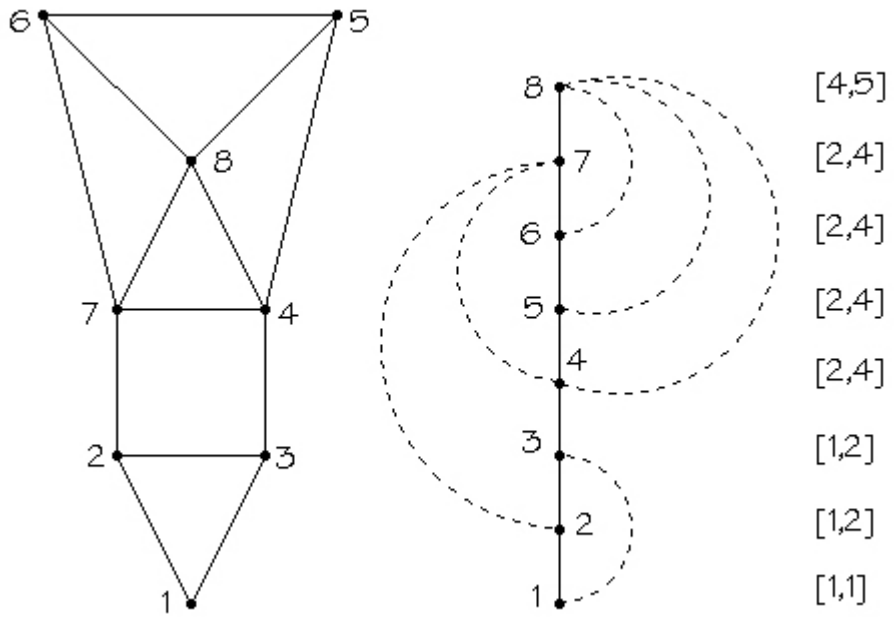

Figure 5.4: Type 2 separation pairs 4,7

and at this point the edge (u, v) will be added to the stack of edges. The procedure stores possible type 2 separation pairs, and continually checks whether or not they satisfy the properties neccessary to be a separation pair.

If a separation pair $\{a, b\}$ is identified, then a virtual edge (a, b, i) (i.e. the ith virtual edge) is added to the stack of edges as well as to the new split component. When the separation pair $\{a, b\}$ is an actual edge in T, a new split component (a triple bond consisting of the virtual edge (a, b, i), the original edge (a, b), and another virtual edge $(a, b, i+1)$) is created. In this latter case the virtual edge $(a, b, i+1)$ is added to the edge stack.

To illustrate the triconnectivity algorithm, consider the biconnected graph $G = (V, E)$ where $\mid V \mid = 11$ and $\mid E \mid = 13$ in the Figure 5.5.

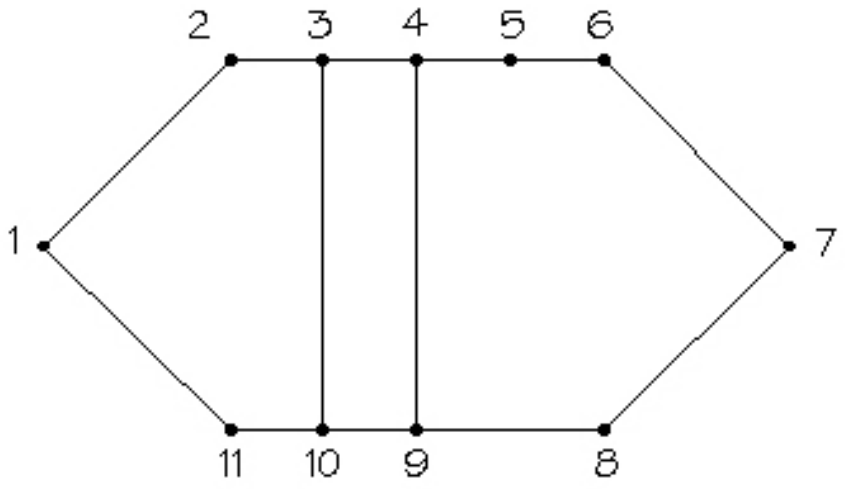

Figure 5.5: G=(V,E), V=11, E=13

The $DFS()$ procedure produces the following set of tree edges and fronds, and corresponding palm tree is shown in Figure 5.6.

$$T \ni (1,2)(2,3)(3,4)(4,5)(5,6)(6,7)(7,8)(8,9)(9,10)(10,11).$$
$$F \ni (9,3)(10,4)(11,1).$$

The $PATHFINDER()$ procedure generates the following set of paths:

$$\{\{1,2,3,4,5,6,7,8,9,10,11,1\},\{10,3\},\{9,4\}\}$$

The $PATHSEARCH()$ procedure then generates the following split components:

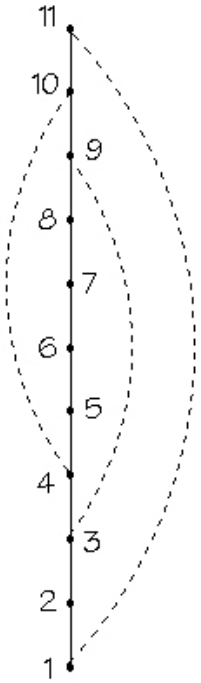

Figure 5.6: Palm tree of graph in Figure 5.5

$\{(1,2)(2,3)(1,3)\}, \{(1,10)(10,11)(1,11)\}, \{(1,3)(1,10)(3,10)\},$
$\{(3,10)(3,10)(3,10)\}, \{(3,4)(3,10)(4,10)\}, \{(4,10)(4,9)(9,10)\},$
$\{(4,9)(4,9)(4,9)\}, \{(4,5)(4,9)(5,9)\}, \{(5,6)(5,9)(6,9)\},$
$\{(6,7)(6,9)(7,9)\}, \{(7,8)(7,9)(8,9)\}$**.**

The separation pairs generated by the algorithm are $\{1,3\}$**,** $\{1,10\}$**,** $\{3,10\}$**,** $\{4,10\}$**,** $\{4,9\}$**,** $\{5,9\}$**,** $\{6,9\}$ **and** $\{7,9\}$**.**

Note that these specific separation pairs and corresponding split components are not unique. Figure 5.8 displays the triconnected components formed from merging the adjacent polygons. The split com-

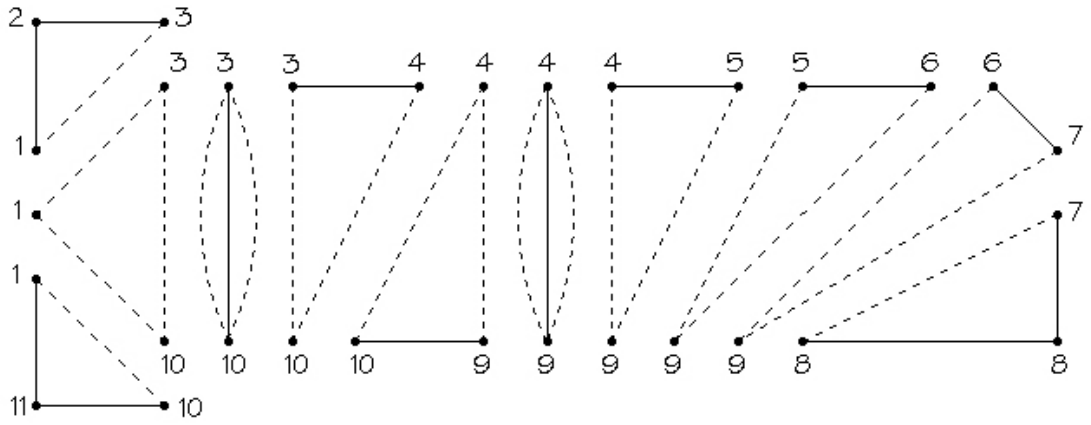

Figure 5.7: Split components of graph in Figure 5.5

ponents {$(3,4)$**,** $(3,10)$**,** $(4,10)$**} and {**$(4,10)$**,** $(4,9)$**,** $(9,10)$**} share a virtual edge** $\{4,10\}$ **and are thus merged to form the biconnected component {**$(3,4)$**,** $(4,9)$**,** $(9,10)$**,** $(3,10)$**}. The split components {**$(1,2)$**,** $(2,3)$**,** $(1,3)$**}, and {**$(1,3)$**,** $(1,10)$**,** $(3,10)$**} share the virtual edge** $(1,3)$ **and the split components {**$(1,3)$**,** $(1,10)$**,** $(3,10)$**} and {**$(1,10)$**,** $(10,11)$**,** $(1,11)$**} share the virtual edge** $(1,10)$**. The three split components are merged together to produce the triconnected component, and {**$(1,2)(2,3)(3,10)(10,11)(11,1)$**}. The split components {**$(4,5)$**,** $(4,9)$**,** $(5,9)$**}, {**$(5,6)$**,** $(5,9)$**,** $(6,9)$**}, {**$(6,7)$**,** $(6,9)$**,** $(7,9)$**}, {** $(7,8)$**,** $(7,9)$**,** $(8,9)$**} are also merged together to form the triconnected component {**$(4,5)$**,** $(5,6)$**,** $(6,7)$**,** $(7,8)$**,** $(8,9)$**,** $(4,9)$**}. Note that the triconnected components in Figure 5.8 consist only of polygons and multilinks and the virtual edges are labeled *a,b,c*, and *d*.**

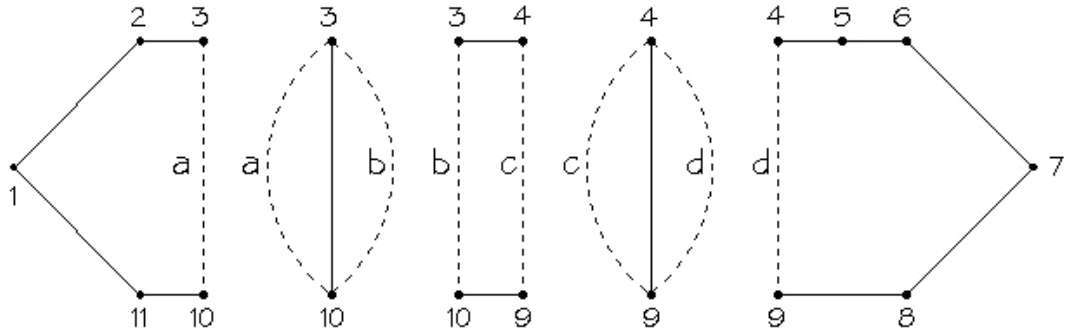

Figure 5.8: Triconnected components of graph in Figure 5.5

Several algorithms exist for drawing triconnected components. In the following section we will consider the multigraph G to be planar. If G is planar then all of the triconnected components of G are planar as well. Given the planar triconnected components (i.e. *3-BlockTree*), the next objective will be to construct a planar representation of each triconnected component. The triconnected components may be embedded individually and there is a unique way of embedding them. Following this will be able to construct a planar representation of the entire multigraph G.

Chapter 6

Graph Drawing Algorithm

Before we can search for dual paths in G, we must first construct a planar embedding of G. The triconnected components contained within the 3-Block Tree serve as building blocks for the planar construction of G. In the following analysis, G is assumed to be planar and therefore each triconnected components of G is planar as well. Recall that the triconnected components are of three types: multilinks, polygons and triconnected graphs. Multilinks consist of two vertices (u, v) with at least three edges incident to u and v. Multilinks can be drawn easily by creating (x, y) coordinates for the two vertices and then drawing the necessary number of edges between them. Polygons are also easy to draw because all that is required is the number of vertices in the polygon. Triconnected graphs are the components of the 3-Block Tree which require the most effort, and a graph drawing algorithm is thus needed. A force-directed placement algorithm presented in [6] and derived from the Fruchterman and Reingold

spring embedding algorithm is used and discussed below.

The graph drawing algorithm works specifically on planar triconnected graphs G, so in the following analysis we assume that G is triconnected and planar. An advantage of this algorithm over others of its type is that besides the vertex and edge set of G, all that is needed to draw G is one face $W = \{w1, w2, w3, ..., wk\} \subset V$ where W is an ordered list of vertices arranged on the face. A face can easily be extracted from the palm tree representation P of G by traversing a sequence of tree arcs followed by one frond. A face can also be identified by running the $PATHFINDING()$ procedure which produces a set of disjoint paths which cover G.

The vertex set V of G is partitioned into two sets, W and $V - W$, where W contains the vertices in the specified face and $V - W$ contains the remaining vertices in V. The algorithm begins by fixing the vertices of the given face $W = \{w1, w2, w3, ..., wk\}$ into a regular polygon of size k inscribed into the unit circle, and centered at the origin. The vertices $v \in W$ are fixed and are not moved during the procedure. The size of the polygon is determined by the number of vertices in the face, and this polygon becomes the outer face of the drawing. The remaining vertices $v \in V - W$ are placed at the origin. The drawing algorithm proceeds to use an iterative pro-

cedure at each step, moving the vertices initially centered at the origin until the entire embedding reaches an equilibrium.

To generate an embedding that reaches an equilibrium, two forces, the *attractive force* $f(u, v)$ and the *resultant force* fr are used. The attractive force is calculated between each pair of adjacent vertices in the graph using a third order law which states that the force exerted by two adjacent vertices u and v on each other is $F_{uv} = Cd^3$. C is a constant and d is the distance between vertices u and v. For each vertex $v \in V - W$, the attractive forces on v are calculated and combined to produce the resultant force of v. At the end of each iteration the vertex v is then moved in the direction of the resultant force. The resultant and attractive forces, and the distance calculated between adjacent vertices all contain two components (x, y) corresponding to the horizontal and vertical directions.

The procedure is executed as follows. At the beginning of each iteration the resultant forces $fr_x(v)$ and $fr_y(v)$ of each vertex $v \in V - W$ are set to 0. For each edge $e = (u, v) \in E$, the attractive forces $f(u, v)_x$ and $f(u, v)_y$ between vertices v and w are determined. Once the forces $f(u, v)_x$ and $f(u, v)_y$ are calculated, they are used to update the resultant forces of vertices u and v, namely $fr_x(u)$, $fr_y(u)$, $fr_x(v)$ and $fr_y(v)$. The attractive force $f(u, v)_x$ (in

the x-direction) is added or subtracted to $fr_x(u)$ and $fr_x(v)$ depending on the relative positions of u and v. For example, if $u_x < v_x$, (the x-ordinate of u is less than the x-ordinate of v), then the attractive force $f(u,v)_x$ is added to $fr_x(u)$ (s.t. u_x is increased *i.e.* **moves horizontally to the right), and the attractive force $f(u,v)_x$ is subtracted from $fr_x(v)$ (s.t. v_x is decreased i.e. moves horizontally to the left). The resultant forces in the y-direction are also updated in the same manner. If $u_y > v_y$, then the attractive force $f(u,v)_y$ is subtracted from $fr_y(u)$ and added to $fr_y(v)$. Once the resultant forces are updated corresponding to each edge $e = (u,v) \in E$, the vertices $v \in V - W$ are moved in the appropriate directions at the end of each iteration.**

The movements are limited, however, by a cooling function $cool(i)$, where i represents the iteration number. The cooling function approaches a limit of 0 as i increases. The cooling function is applied to the procedure due to the fact that if the displacements at each step are too large then the algorithm may require a large number of iterations to reach an equilibrium. Thus to ensure that the drawing reaches an equilibrium, and thus a planar embedding in a relatively small number of steps, the vertices are allowed to move in the directions proscribed by the resultant force, but only in an amount proportional to the iteration number dic-

tated by the cooling function.

The graph drawing algorithm runs for a number of iterations specified initially. Recommended values for the constant C, used to calculate the attractive forces, and the cooling function $cool(i)$ are specified in [6] below where n is the number of vertices in V and i is the iteration number:

$$C = \sqrt{\frac{n}{\pi}} \; cool(i) = \frac{\sqrt{\frac{\pi}{n}}}{1+\frac{\pi}{n}i^{3/2}}$$

An illustration of the graph drawing algorithm is presented below. Consider the following palm tree P of the triconnected graph G with $|V| = 10$ and $|E| = 21$ in Figure 6.1.

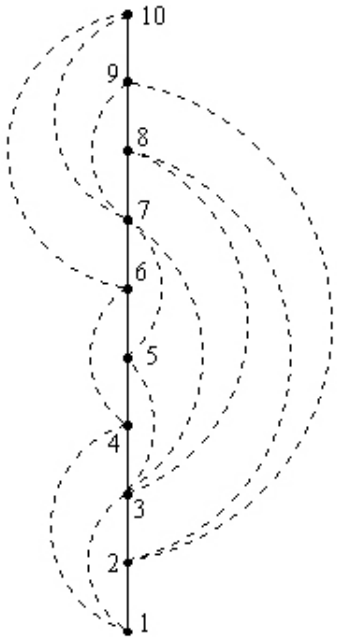

Figure 6.1: Palm tree P of G

Suppose we choose the face $W = \{1, 4, 6, 10, 9, 2\}$ to be the outer face. This face has 6 vertices so we must inscribe these 6 vertices in a hexagon. The interior angle formed by adjacent edges in the polygon is found by the formula $(2\pi/k)$ where k is the number of vertices in the outer face, and in this example equals $60°$.

The radius is always set to 1 as to to inscribe the face vertices into the unit circle. The coordinates of the vertex $w_n \in W$ are calculated by $w_n(x, y) = (cos(angle{*}n), sin(angle{*}n))$ where n is the index number of the vertex in $W = \{w_0, w_1, w_2, ...w_{k-1}\}$.

The following figures display four separate iterations of the algorithm. Figure 6.2 displays the initial positions of all the vertices in G. The vertices of W are inscribed in the hexagon and the remaining vertices are positioned at the origin. Figure 6.3 shows the embedding after 1 iteration, and Figures 6.4 and 6.5 display the embeddings of G after 2 and 5 iterations respectively. The algorithm produces a planar embedding after 1 iteration, and the remaining iterations only improve upon the initial embedding slightly.

Once each triconnected component has its planar representation, the next step is to combine them into one planar representation. Polygons and multilinks as discussed previously are trivial to draw. The separation pairs of a biconnected graph are

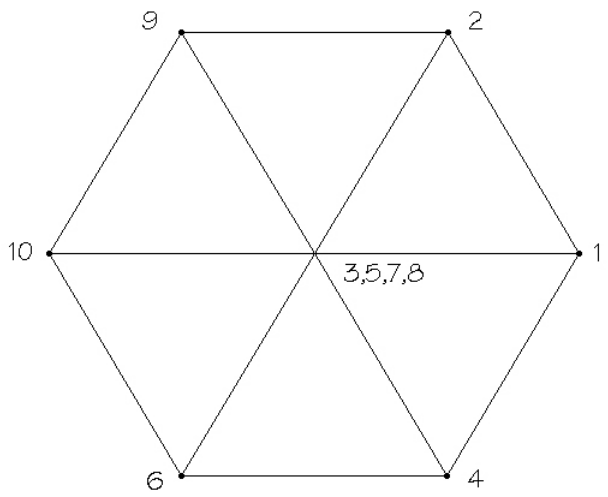

Figure 6.2: Iterations = 0

used as a guide to merging the individual planar representations of the triconnected components. A recursive algorithm is used to merge the planar representations of the triconnected components into one planar biconnected component.

During the construction of the 3-Block Tree, each triconnected component is numbered, and following the construction the algorithm determines which triconnected components are adjacent by identifying triconnected components which share a virtual edge. The drawing algorithm creates a final embedding, $Final[]$, which combines all the embeddings of the triconnected components. The algorithm uses a depth-first search type procedure to combine the individual embeddings, beginning with the triconnected component numbered 0. The embedding of

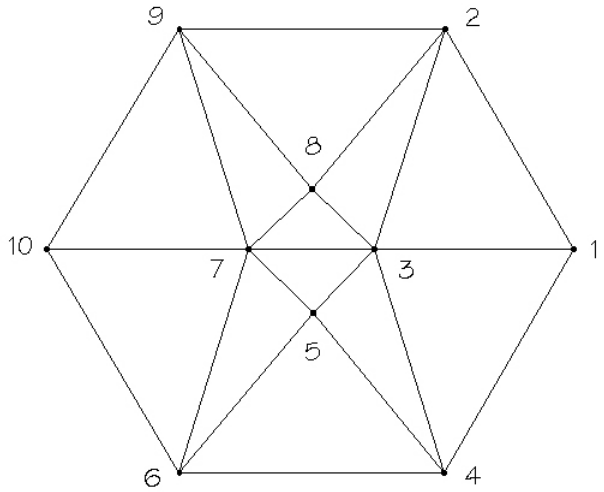

Figure 6.3: Iterations = 1

component 0 is initially stored into $Final[]$. The algorithm then identifies the triconnected component adjacent to triconnected component number 0 and merges their planar representations. The embeddings are merged such that the components are drawn on opposite sides of the separation pair. The merge is accomplished through updating $Final[]$ to include the coordinates of the vertices from the adjacent triconnected component.

Consider the example in Figure 6.6. There are 3 triconnected components numbered $1, 2$, and 3, and each component has its own planar embedding. The coordinates of the vertices in component 1 are initially stored into $Final[]$. Components 1 and 2 share the virtual edge $(b, e, 1)$. Since components 1 and 2 are adjacent, the embedding of component 2 must

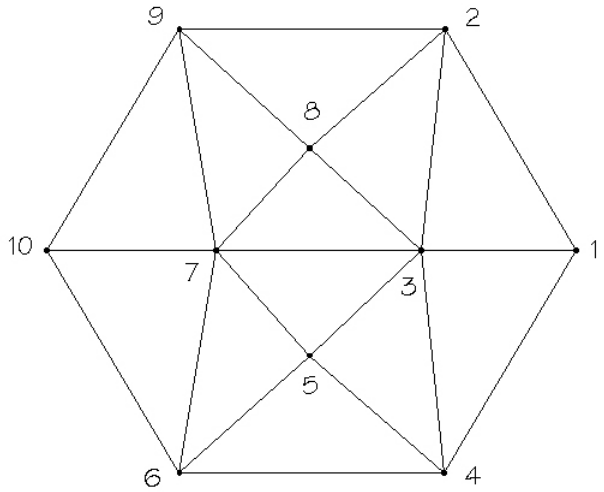

Figure 6.4: Iterations = 2

be merged with $Final[]$, since $Final$ now contains the embedding of component 1.

The embeddings of $Final$ and component 1 are merged using the following procedure. $Final[]$ is first rotated such that the virtual edge $(b, e, 1)$ is parallel to the horizontal axis, and such that all other vertices in $Final[]$, specifically a and f, are positioned below the shared virtual edge $(b, e, 1)$. Component 2 is rotated s.t. $(b, e, 1)$ is parallel to the horizontal axis and all other vertices are positioned above the shared virtual edge $(b, e, 1)$. The distance between b and e of component 2 is set to the same distance between b and e of $Final[]$. A check is then performed to determine whether or not the vertices b and e of component 1 are positioned in the same order (i.e. b is to the left or right of e) as b and e in $Final[]$. If

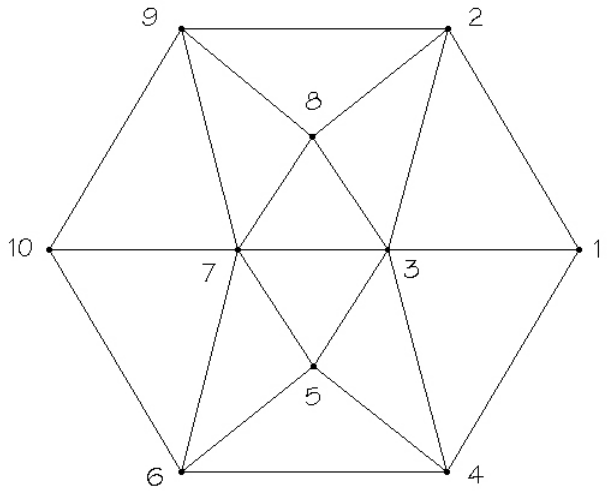

Figure 6.5: Iterations = 5

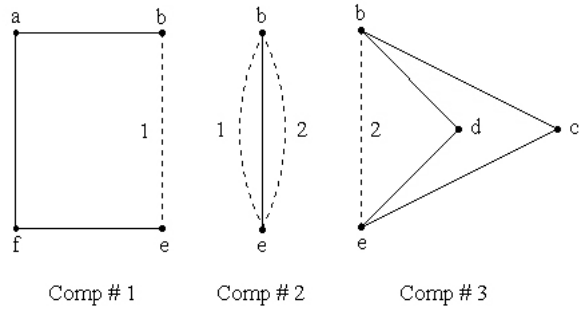

Figure 6.6: Triconnected components of G

positions of b and e of the component to be merged are in a different order, the positions are reflected about the x-axis such that their resulting order corresponds to the order of b and e in $Final$. Finally, the positions of all the vertices in component 2 are translated such that the coordinates of a and b in component 2 are equal to the coordinates of a and b in $Final[]$. Now $Final[]$ contains the embeddings

of components 1 and 2, and the algorithm proceeds to merge the remaining triconnected components in the same manner. After all the triconnected components are merged into $Final[]$, the algorithm then draws edges between adjacent vertices. The final result is a planar embedding of the entire biconnected graph. The planar embedding generated from the triconnected components in Figure 6.6 is shown in Figure 6.7.

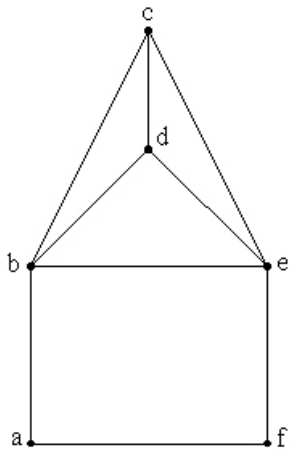

Figure 6.7: Final planar embedding of G

The algorithm uses several procedures to manipulate the positions of the embeddings. These procedures include rotating a set of vertices by a given angle, translating a set of vertices a certain distance in the horizontal and vertical directions, reflecting a set of vertices about the vertical axis $(x = 0)$, and determining the interior angle formed between an edge and the horizontal axis $(y = 0)$. A set of ver-

tices are rotated a certain angle using a *rotation matrix* as follows. In order to rotate each vertex θ degrees, each pair of coordinates is set

$$v_x = v_x \cdot \cos(\theta) + v_y \cdot \sin(\theta)$$
$$v_y = -v_x \cdot \sin(\theta) + v_y \cdot \cos(\theta)$$

Vertices are translated x units in the horizontal direction and y units in the vertical directions by simply increasing the x-ordinate of each vertex by x units and increasing the y-ordinate of each vertex by y units. Vertices are reflected about the vertical axis $(x = 0)$ by multiplying the x-ordinate of each vertex by -1. The Dual-Eulerian section will discuss additional procedures which are used to calculate the angle between an edge and the horizontal axis, and the angle between two edges in G.

The procedure presented in this section creates from the 3-Block Tree of a graph G a planar embedding of G. The procedure assumes that G is planar. The graph drawing algorithm for triconnected graphs has previously been implemented in a Mathematica-based system called Vega[6]. There are no other implementations of the procedure for merging individual embeddings of triconnected components into a single planar biconnected multigraph. The following section presents the algorithm for finding dual paths of a plane multigraph.

Chapter 7

Dual-Eulerian Algorithm

Several algorithms exist for determining whether or not a biconnected plane multigraph is Dual-Eulerian. This section presents an algorithm presented in [9] for identifying Dual-Eulerian trails in biconnected plane multigraphs and consequently for determining whether or not a plane multigraph is Dual-Eulerian. Before presenting the algorithm lets first discuss the related mathematical concepts.

A *Petrie* tour (walk) in a plane multigraph is a tour (walk) which turns alternatingly left and right. Often Petrie walks are indicated by representing the turns made at each vertex by small arcs. Figure 7.1 displays a right turn, a left turn, and a Petrie walk. The reverse of a Petrie walk is also a Petrie walk. Petrie walks have the interesting property that the same Petrie walk in G is also a Petrie walk in G^*. *Euler-Petrie* tours are simply Eulerian Petrie tours which by the definition of Eulerian tours contain every edge e in G. Therefore, since Petrie tours (walks) correspond to dual tours

(walks), an Euler-Petrie tour (walk) is also a dual Eulerian tour (walk).

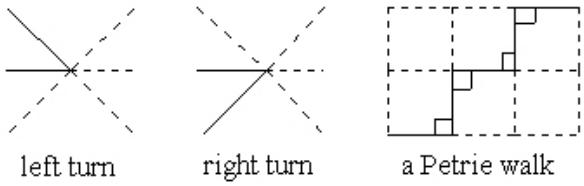

Figure 7.1: Turns and walks

Key ideas regarding Petrie trails in plane graphs which are used as the basis for this algorithm are presented in [9]. If a plane multigraph G has a Dual-Eulerian tour t, then G also has a Dual-Eulerian Petrie tour $t\prime$ which is also Petrie. We are primarily concerned with identifying Dual-Eulerian tours and walks in biconnected plane multigraphs, and for these multigraphs an even more interesting property is identified. Namely, if a biconnected plane multigraph G has a dual Eulerian tour t, then t is a petrie trail. The problem of finding dual paths in biconnected plane multigraphs can be simplified to finding Petrie trails in either G or G^*.

Previous algorithms exist for determining whether or not a biconnected plane multigraph is Dual-Eulerian through identifying maximum size dual trails. The algorithm in [1] requires as input the planar embedding of both G and G^*. Each edge e in G and G^* is ordered and the edge successors (edges which follow

(u, v)) are identified. A dual successor of an edge e in G and G^* is an edge successor of e in both G and G^*. Dual trails are then constructed from these dual successors. Given an embedding of a plane multigraph, the algorithm implemented in this study requires considerably less work than the algorithms which use the methods just described.

Given the biconnected planar embedding of G, which can be constructed from the 3-Block Tree of G, we are ready employ our algorithm to search for dual trails. The objective is to find the maximum size Petrie walk in a biconnected graph G. The algorithm discussed here terminates once the maximum size Petrie walk is found. If the generated Petrie walk is Eulerian then the Petrie walk is Dual-Eulerian and G is therefore Dual-Eulerian. We can search for dual trails by starting from any edge (u, v) and proceed to alternate turning left and right until an edge is traversed twice. If this proves unsuccessful, i.e. we don't construct an Euler-Petrie trail, we start from the same edge and begin turning right-left, right-left. Note that starting from any edge (u, v), there are two directions we may move. Either from $u \to v$, in which case the turn is made at vertex v, or from from $v \to u$, in which case the turn is made at vertex u. We also have two options for the starting turn (right or left). Once the direction of the starting edge and the initial turn is specified,

we are ready to begin.

The Petrie walk p is constructed until an attempt is made to add an edge of G already in the Petrie walk p. If p is constructed such that no edge is repeated and every edge of G is in p then the algorithm terminates successfully, and G is Dual-Eulerian. If the construction attempts to add an edge e to p, where e is already contained in p and there exists edges in G that have not been added, then the procedure pauses. In this scenario the Petrie walk p cannot be extended any farther from edge e. The Petrie walk may, however, be able to extend from the starting edge e going in the opposite initial direction making the same initial turn. Any edges that are added in this manner will increase the size of the Petrie walk p. Once an edge is approached which has already been added, the algorithm then terminates. If the generated Petrie walk is not Eulerian then we repeat the entire procedure starting from the same starting edge e but with the opposite initial turn.

Consider the following biconnected plane graph G where the vertices are numbered 1 through 6 and the edges are labeled a through h. Our aim is to construct the largest size Petrie walk starting from edge a. Starting from edge a we can either turn left or right at vertex 1, or we can turn left or right at vertex 2. Turning from edge a at vertex 2, a

right turn will lead to edge g, and a left turn will lead to edge b. Suppose we turn right onto edge g. Our Petrie walk is now $a \to g$. We must now alternate turns, so we will turn left from edge g. Turning left from edge g leads to edge f, resulting in the Petrie walk of $a \to g \to f$. At edge f we must now turn right onto edge d, resulting in the Petrie walk of $a \to g \to f \to d$. At edge d we must now turn left onto edge c, resulting in the Petrie walk $a \to g \to f \to d \to c$. From edge c we must now turn right onto edge b, resulting in the Petrie walk $a \to g \to f \to d \to c \to b$. Now from edge b we must turn left onto edge g. This creates a problem because edge g is already in our Petrie walk. Figure 7.2 displays this construction where the Petrie walk is indicated by dotted lines.

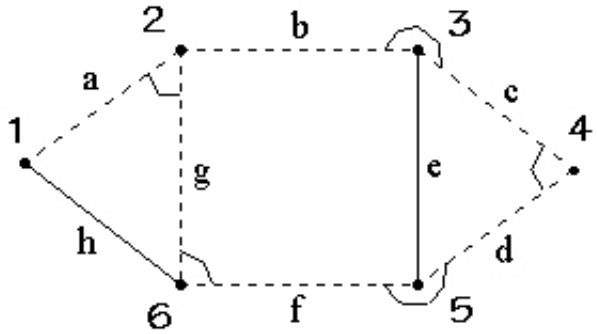

Figure 7.2: Petrie walk {a,g,f,d,c,b}

We therefore save our construction and go back to the starting edge a. Initially we traversed a from

$1 \to 2$, **turning right at vertex 2. So now we switch directions and turn right at vertex 1. Turning right from edge a at vertex 1 leads to edge h. We must turn left from edge h, but doing so leads to edge g which is already in our Petrie walk. We now can conclude than our particular Petrie walk $h \to a \to g \to f \to d \to c \to b$ is maximal. Figure 7.3 shows the last step of the construction of adding on edge h.**

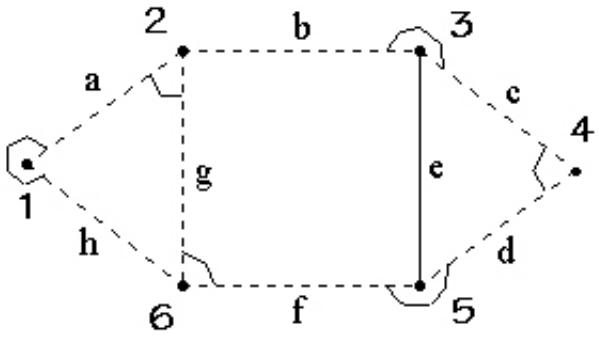

Figure 7.3: Petrie walk {h,a,g,f,d,c,b}

The Petrie walk is not Eulerian, however, therefore we must repeat the procedure turning left initially from edge a in the same initial direction $1 \to 2$. The Petrie walk constructed from turning left at edge a is $a \to b \to e \to d \to c$. At edge c we proceed to turn left but doing so leads to edge e which has already been added to the Petrie walk. This Petrie walk is displayed in Figure 7.4.

We again save our construction and go back to the starting edge a. Initially we traversed a from $1 \to 2$

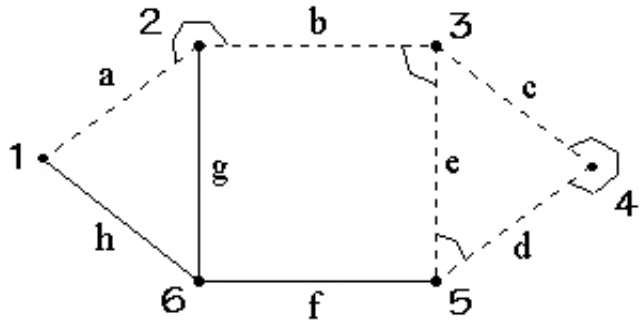

Figure 7.4: Petrie walk {a,b,e,d,c}

turning left at vertex 2, so now we switch directions and turn left at vertex 1. Turning left from edge a at vertex 1 leads to edge h. We must now turn right from edge h at vertex 6, which leads to edge f. Now we must turn left from edge f at vertex 5, but doing so leads to edge e which is already in our Petrie walk. We can thus conclude than our new Petrie walk $f \to h \to a \to b \to e \to d \to c$ is maximal. Figure 7.5 shows this final construction.

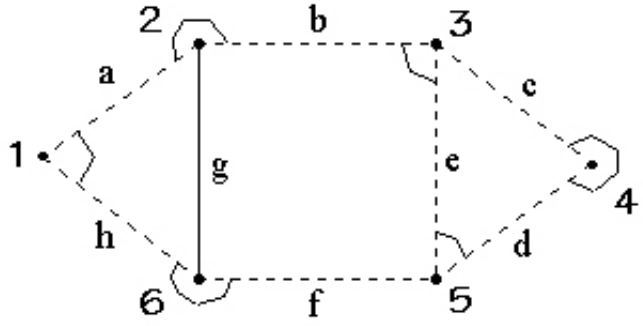

Figure 7.5: Petrie walk {f,h,a,b,e,d,c}

In the previous example both Petrie walks were not Eulerian and they both contained 7 edges. Based on this fact we can conclude that the given biconnected graph G is not Dual-Eulerian. This method for finding Dual-Eulerian paths can be extended to finding the minimum number of dual paths which cover a biconnected graph. Once a maximal size Petrie walk is created, the next step is to search for Petrie walks in the set of edges not contained in the initial Petrie walk This general method is used in conjunction with the 3-Block Tree of a given biconnected graph to solve the more difficult problem of determining whether a planar multigraph is dual Eulerian. The algorithm in [8] analyzes the topologies of Euler-Petrie paths as they cross back and forth between the separation pairs of a multigraph without considering specific planar embeddings.

The implementation of the Dual-Eulerian algorithm presented above uses some of the same procedures as the graph drawing algorithm. The graph is rotated at each step of the construction in order to determine which edges correspond to right and left turns. From any edge $e = (a, b)$, the right and left turns made at vertex b are determined by calculating the interior angles formed by e and all edges adjacent to e which have an endpoint at vertex b. To accomplish this the graph is shifted such that vertex a is positioned at the origin. The angle θ

between vertex b and the horizontal axis is then determined, and then the graph is rotated by θ such that edge e is parallel to the horizontal axis. The interior angles formed by the edges adjacent to e having b as an endpoint are then calculated using the *law of cosines*.

Consider the graph in Figure 7.6. Suppose the edge $e = (a, b)$ is under consideration. To determine the interior angle α formed between edge (a, b) and edge (b, c), we first calculate the three distances ab, bc, and ac using the distance formula $d = \sqrt{(x_2 - x_1) + (y_2 - y_1)}$. The angle α is then calculated using the law of cosines by the following formula:

$$\alpha = \arccos(\tfrac{ac^2 - ab^2 - bc^2}{-2(ab)(bc)}).$$

Once the interior angle of each edge $f = (b, c)$ adjacent to $e = (a, b)$ is determined, the algorithm checks to see whether or not the y-coordinate of c is less than the y-coordinate of b. If it is, then the angle α calculated from the previous formula must be subtracted from 360. Once all the angles formed by edge e and each edge adjacent to e are determined, the adjacent edge that forms the minimum angle corresponds to a left turn and the adjacent edge that forms the maximum angle corresponds to a right turn. This procedure is capable of determining the left and right turns of any edge in the graph in either direction.

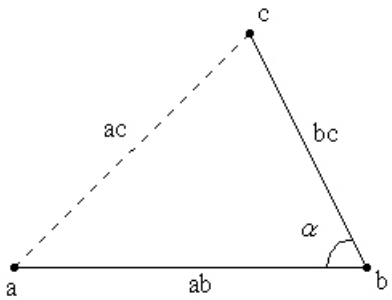

Figure 7.6: Calculating the angle α between two edges

The Dual-Eulerian algorithm constructs Petrie trails by keeping track of each turn and each edge which is added to the Petrie trail. A check is performed at each step to verify that the edge being added is not already in the Petrie trail. The number of steps required to translate the graph, rotate the graph, and calculate angles between adjacent edges is quadratic with respect to the number of vertices and edges in the multigraph. There are no other working implementations of the Dual-Eulerian algorithm presented in [9].

Chapter 8

Conclusions and Further Work

The implementation of the triconnectivity, graph drawing and Dual-Eulerian algorithm presented in this paper successfully determines in polynomial time whether or not a plane multigraph is Dual-Eulerian. The implementation presented in this paper was tested on several multigraphs ranging from 10 to 100 vertices. The entire algorithm was implemented in Borland C++, and the computer code is displayed in the Appendix.

Further work can be done on the implementation by adding a few key procedures. The planarity algorithm discussed in the depth-first search section is a natural fit because this implementation uses the same data structures. The implementation presented in this paper assumes the multigraph is planar, but for the implementation to be complete, a planarity check is necessary. The implementation presented in this paper identifies whether or not a plane multigraph is Dual-Eulerian and finds the Dual-Eulerian trail if it exits, but it doesn't

find the minimum number of dual trails that cover the multigraph. Again, the implementation possesses the necessary data structures and functions to tackle this problem.

Lastly, the implementation can be extended to answer the more difficult question of whether or not a multigraph admits an embedding which is Dual-Eulerian, or can be extended to find the embedding for which the minimum number of dual trails exist. Either one of these latter problems would be of the most interest to designers of VLSI circuits.

Chapter 9

Implementation

```
/* Triconnectivity Procedure
 Input: Biconnected planar graph
 Output: Planar representation and determination
         of whether or not G is Dual-Eulerian
 This program contains the triconnected components algori
 the procedure for drawing the triconnected components, t
 for merging the individual planar embeddings, and the pr
 finding Petrie trails in the planar embedding of G. The
 components algorithm is from [4], the graph drawing algo
 and the Dual-Eulerian algorithm is from [9].
*/

/* The implemenatation accepts graphs in the form of adja
   used in main(). The implementation accepts adjacency s
   each list is in increasing order.
*/


#include <vcl.h>
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>

#include "structures.h"

#define START 1000
#define EOS 100
#define TRUE 1
#define FALSE 0
#define PI 3.141592654
#define TOL .00001
#define TOL2 .001
#define LEFT 0
#define RIGHT 1

int E=0,V=0,nodes=0,m=0;   // Variables for the number of
int num_comp=0;            // Number of split components
double radius = 2.0;       // Radius of polygons to be in
int iterations = 15;       // Iterations for graph drawin


Pair Fr[15];     // Resultant Force ..
Pair Fa;         // Force between adjacent vertices ...

EDGE SepPair[50];          // Separation Pairs
```

```
Component C[50], Final;    // Split Components ..
Bucket BUCKET[50];         // 2V + 1 buckets ..
Triple TSTACK[50];         // Possible Type 2 separation
EDGES ESTACK[50];          // Stack of edges
Triple Last_Deleted;       // Last type 2 separation pair
Adj_List A[14], A_1[14];   // Adjacency Lists
PATH Paths[14];            // Store paths from PATHFINDER

EDGES Comp_A[15][15];  // Split Components adjacency stru
int Comp_D[15];        // Degree of Split Components (Use

int v_cnt = 0; int v_one = 0; int c = 0;

int theta[13][13];   // theta(e) - used to create new adj
int MARK[20][20];
int curr_a=0, curr_h=0, curr_b= 0, curr_x=0, curr_y=0, cu
int v1, w1, n, NUMBER[14], FLAG[14], Edge[14][14];
int T_number[30];
int LOWPT1[14], LOWPT2[14], ND[14], FATHER[14], FLAG1, j=
int DEGREE[14], A1[14];
int NEWNUM[14], HIGHPT[14], s, path=0, path_cnt;
int tstack = 0, estack=0, numcomp=0, seppair=0, vedges=0;
int global_check,r=0;
int m_edge;      // Multiple edge # ....

int SPECIAL[30], STAY=0;
int DUAL=0;
double dx=0.0,dy=0.0,hyp=0.0;
```

```
int turn=0,stack_cnt=0,START_dir=0;
int W[30][30];

Pair C1[30],C2[30],Temp[30],Tr[30];
Direction DIR;
EDGE STACK[30],Strt,Prev;
VIRT Vrt[30];

void DFS(int v, int u);
void PATHFINDER(int v);
void PATHSEARCH(int v);
int PATHCHECK(int v, int x);

void DELETEA(int h, int a, int b);
void ADD(int h, int a, int b);
void DELETE_E(int u, int v, int z);
void ADD_E(int u, int v, int z);

void Draw_T(int i);
double cool(int i);
double rounder(double r);

void Translate_C(int t, double shiftx,double shifty);
void Rotate_C(int t, double theta);
double Get_Angle_X_Axis_C(int t,int u, int v);

void Fix(int t,double d);
void Flip(int t);
```

```
void Translate_F(double shiftx,double shifty);
void Rotate_F(double theta);
double Get_Angle_X_Axis_F(int u, int v);

void Mult_Edge(int comp, int u, int v, int direction);

void Merge_3_Block_Tree(int cnt);
void Merge_Pos_E(int s);
void DFS_T(int v);

void Create_Adj(void);
void Merge_Poly(void);

void Print_C2(void);
void TURN(int a,int b,int c);
int STACK_CHECK(int b,int next);
void Translate(double shiftx,double shifty);
void Rotate(double theta);
void RESET(void);
double Get_Angle_X_Axis(int u, int v);

void Acc_Adj(void);    // Create new adjacency atructure

void Translate(double shiftx,double shifty){

  // Translate shiftx in x-direction and shifty in y-dire
```

```
  for(int i=1;i<(nodes+1);i++){
    C2[i].x = C1[i].x + shiftx;
    C2[i].y = C1[i].y + shifty;
  }

}

void RESET(void){    // Reset coordinates in C2[] to C[].

  for(int i=1;i<(nodes+1);i++){
    C2[i].x = C1[i].x;
    C2[i].y = C1[i].y;
  }
}

void Rotate(double theta){  // Rotate coordinates in C2[]

  for(int i=1;i<(nodes+1);i++){

    // Change angle in degrees to radians ..

    Temp[i].x = C2[i].x*cos(theta*PI/180)
                + C2[i].y*sin(theta*PI/180);
    Temp[i].y = -C2[i].x*sin(theta*PI/180)
                + C2[i].y*cos(theta*PI/180);

    // Round coordinates to nearest integer ..
```

```
    if(fabs(Temp[i].x) < TOL) Temp[i].x = 0;
    if(fabs(Temp[i].y) < TOL) Temp[i].y = 0;
    if(fabs(Temp[i].x - ceil(Temp[i].x)) < TOL2 )
              Temp[i].x = ceil(Temp[i].x);
    if(fabs(Temp[i].y - ceil(Temp[i].y)) < TOL2 )
              Temp[i].y = ceil(Temp[i].y);

  }

  // Store in Coordinate lists ...

  for(int i=1;i<(nodes+1);i++){
    C2[i].x = Temp[i].x;
    C2[i].y = Temp[i].y;
  }

}

int STACK_CHECK(int b,int next){
// Check if (b,next) is on stack in the Dual-Eulerian alg
// Procedure checks if edge is already in Petrie trail

  int check=0,next_turn=0;

  for(int i=0;i<stack_cnt;i++)
    if(  ((STACK[i].u==b) && (STACK[i].v==next)) ||
 ((STACK[i].u==next) && (STACK[i].v==b)) ) {
```

```
      if(stack_cnt < E){

        check = 1;
cout << "(" << b << "," << next << ")" << endl;
cout << "stack_count = " << stack_cnt << endl;
cout << "Problem - (" << b << "," << next
             << ") already on stack ..." << endl;

      }

      else {

check = 1;

cout << "(" << b << "," << next << ")" << endl;
cout << "stack_count = " << stack_cnt << endl;
cout << "(" << b << "," << next
             << ") is the next edge" << endl;
cout << "(" << turn
             << ") is the previous direction ... (L=0,R=1

if(turn==LEFT) next_turn=RIGHT;
else next_turn=LEFT;

cout << "(" << next_turn
             << ") is the next direction ... (L=0,R=1)" <
cout << " .... (" << Strt.u << "," << Strt.v
             << ") is the START edge" << endl;
```

```
cout << " .... (" << START_dir
             << ") is the START direction ... (L=0,R=1)"

if( (Strt.u==b) && (Strt.v==next) && (START_dir==next_tur
  cout << "DUAL-EULERIAN" << endl;
  DUAL = 1;

      }

    }

  return(check);

}


double Get_Angle_X_Axis(int a, int b){

    double theta = 0.0;
    // Calculate angle from pos. x-axis to b ..
    // Distances are created from b(x,y) - a(x,y) ...

    dx = C2[b].x - C2[a].x; if(fabs(dx) < TOL) dx = 0;
    dy = C2[b].y - C2[a].y; if(fabs(dy) < TOL) dy = 0;

    if((dx>0)&&(dy==0))     // In position, don't rotate
        theta = 0;
```

```
    if((dx != 0)&&(dy > 0)){          // Quadrant I & Qua
       theta = (180.0/double(PI))*acos(dx/sqrt(pow(dx,2)+
    }
    else if((dx != 0)&&(dy < 0)){     // Quadrant III & Q
       theta = 360 - (180.0/double(PI))*acos(dx/sqrt(pow(
    }
    else if((dx==0)&&(dy > 0)){  // Vertical Line b_y > a
       theta = 90;
    }
    else if((dx==0)&&(dy < 0)){  // Vertical Line a_y > b
       theta = 270;
    }
    else if((dx < 0)&&(dy==0)){  // Horizontal Line b_x <
       theta = 180;
    }

    // cout << "theta = " << theta << endl;
    return(theta);

}

void TURN(int a,int b, int turn){

  // Input: Edge (a,b)
  // Output: The right and left turns from edge (a,b) at

  double temp=0,min=0,max=0;
  double tx=0,ty=0,t11=0,slope=0;
```

```
  int right=0,left=0,c=0,j=0;
  double angle=0,theta=0,theta1=0;

  DIR.left=0;
  DIR.right=0;

  // cout << " ..... (" << a << "," << b << "," << turn

  if(A[b].deg == 2) {                    // if A[b].deg ==

    for(j=0;j<A[b].deg;j++)              // For each neighb
      if(A[b].neigh[j] != a)             // if c != a ...
right = left = A[b].neigh[j];

       // cout << " ... right = left = " << right << end

   }

  else {             // A[b].deg > 2 ...

    RESET();

    // Shift a to origin ...
    Translate(-C2[a].x,-C2[a].y);

    // Calculate angle between b and x-axis .. (theta)
    theta = Get_Angle_X_Axis(a,b);
```

```
// Rotate s.t. (a,b) || to x-axis ...
Rotate(theta);

// Find angle between a-b and all it's neighbors c  .

min = 400;
max = 0;

for(j=0;j<A[b].deg;j++) {           // For each neigh

   c = A[b].neigh[j];

   if(c!=a) {          // if c != a ...

   // Calculate |ab|, |bc|, |ac| and b_y - c_y ....

   double ab = sqrt( pow(C2[a].x-C2[b].x,2) + pow(C2[
   double bc = sqrt( pow(C2[b].x-C2[c].x,2) + pow(C2[
   double ac = sqrt( pow(C2[a].x-C2[c].x,2) + pow(C2[
   double d_bc_y = C2[b].y - C2[c].y;

   temp =  ( pow(ac,2) - pow(bc,2) - pow(ab,2) ) / (-

   theta = 0;       // Reset theta ...

   if( (abs(temp) < 1) || (abs(temp) == 1) ) {
      angle = acos(temp);
      theta = (180.0/double(PI))*angle;
```

```
        } else cout << "Problem in calculation ...." << en

        if(d_bc_y > 0) theta = 360 - theta;

        // cout << a << "-" << b << "-" << c << " ... thet
        // getchar();

        if(theta < min) {
  min = theta;
  left = c;
          // cout << "min= " << min << endl;
}

if(theta > max) {
  max = theta;
  right = c;
          // cout << "max= " << max << endl;
}

         }  // end if()
        }   // end for()
    }   // end else()

  // cout << "LEFT-" << left << " ... RIGHT-" << right <<

  DIR.left = left;
  DIR.right = right;
```

```
}

void Print_C2(void){      // Print coordinates in C2[] ..

  cout << "----------- Coordinates -------------" << endl
  for(int i=1;i<(nodes+1);i++)
    cout << i << ": {" << C2[i].x << "," << C2[i].y << "}

}

void Flip(int t){

  // Flip entire component about line (x=0) ..

  for(j=0;j<C[t].face_v;j++){

    C[t].Pos[ C[t].F[j] ].x = -1 * C[t].Pos[ C[t].F[j] ].

    // Round coordinates to nearest integer ..
    if(fabs( C[t].Pos[C[t].F[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].x - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].x = ceil(C[t].Pos[C[t].F[j]].x)
    if(fabs( C[t].Pos[C[t].F[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].y = ceil(C[t].Pos[C[t].F[j]].y)
  }

  for(j=0;j<C[t].inner_v;j++){
```

```
    C[t].Pos[ C[t].I[j] ].x = -1 * C[t].Pos[ C[t].I[j] ].

    if(fabs( C[t].Pos[C[t].I[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].x - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].I[j]].x = ceil(C[t].Pos[C[t].I[j]].x)
    if(fabs( C[t].Pos[C[t].I[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].I[j]].y = ceil(C[t].Pos[C[t].I[j]].y)
  }
}

void Fix(int t,double d){

  // Make positions of vertices of the separation pair
  // identical to the positions in Final().
  // Move each vertex in d/2 units either left or right .
  // d = df - ds ....

  // cout << "FIX " << (d/2.0) << endl;

  for(int j=0;j<C[t].face_v;j++){

    if(C[t].Pos[ C[t].F[j] ].x <= 0)
      C[t].Pos[ C[t].F[j] ].x = C[t].Pos[ C[t].F[j] ].x -
    else
      C[t].Pos[ C[t].F[j] ].x = C[t].Pos[ C[t].F[j] ].x +
```

```
    // Round coordinates to nearest integer ..
    if(fabs( C[t].Pos[C[t].F[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].x - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].x = ceil(C[t].Pos[C[t].F[j]].x)
    if(fabs( C[t].Pos[C[t].F[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].y = ceil(C[t].Pos[C[t].F[j]].y)
  }

  for(j=0;j<C[t].inner_v;j++){

    if(C[t].Pos[ C[t].I[j] ].x <= 0)
      C[t].Pos[ C[t].I[j] ].x = C[t].Pos[ C[t].I[j] ].x -
    else
      C[t].Pos[ C[t].I[j] ].x = C[t].Pos[ C[t].I[j] ].x +

    if(fabs( C[t].Pos[C[t].I[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].x - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].I[j]].x = ceil(C[t].Pos[C[t].I[j]].x)
    if(fabs( C[t].Pos[C[t].I[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].I[j]].y = ceil(C[t].Pos[C[t].I[j]].y)
  }
}


void Merge_Pos_E(int s){
```

```
int k=0,j=0,fpos=0,spos=0;

// Check for bond type ....

if( (strcmp(Type[C[s].type],"BONDS") != 0) ){

// Merge positions of C[s] and Final[] into Final[] ...
// Position Final[] s.t (a,b,i) is on top ...
// Position C[s] s.t. (a,b,i) is on bottom ...
// Then store results in Final[] .....
// Shared edge (a,b,i) ........

// cout << "Merge Positions of C[" << s << "] w/ Final[

// Find shared virtual edge of Final[] and C[s] ..
for(k=0;k<Final.e;k++)
   for(j=0;j<C[s].e;j++)
     if( (Final.E[k].u == C[s].E[j].u)  && (Final.E[k].
     // share a virtual edge ..
         fpos = k; spos = j;
       }

// Get Angle between (y=0) and v from shared edge (u,v)
double theta = Get_Angle_X_Axis_F(Final.E[ fpos ].u, Fi

// Rotate s.t. (a,b) is || to x-axis ...
Rotate_F(theta);
```

```
// Check if (a,b) on top using any other vertex from th
int check=0;
for(k=0;k<Final.e;k++)
   if( ( Final.Pos[ Final.E[k].u ].y > Final.Pos[ Final
       ( Final.Pos[ Final.E[k].v ].y > Final.Pos[ Final
     check=1;
if(check == 1) Rotate_F(180.0);    // rotate s.t (a,b)

// Now set the positions of the component to be merged
// corresponding positions of Final()

theta = Get_Angle_X_Axis_C(s,C[s].E[spos].u,C[s].E[spos

// Rotate s.t. (a,b) is || to x-axis ..
Rotate_C(s,theta);

// Check if (a,b) on top using face vertices ...
check=0;
for(k=0;k<C[s].face_v;k++)
   if( (C[s].F[k] !=  C[s].E[ spos ].u) && (C[s].F[k] !
     if( ( C[s].Pos[ C[s].F[k] ].y < C[s].Pos[ C[s].E[
       ( C[s].Pos[ C[s].F[k] ].y < C[s].Pos[ C[s].E[ sp
       {

        check=1;
       }

// getchar();
```

```
if(check == 1) Rotate_C(s,180.0);    // rotate s.t (a,b

// Check if edge size is equal, if not Fix --- --------
double ds = fabs( C[s].Pos[ C[s].E[ spos ].u ].x - C[s]
double df = fabs( Final.Pos[ Final.E[ fpos ].u ].x - Fi
if( rounder(df-ds) != 0.0) Fix(s,df-ds);

// Check alignment, if the order of (u,v) is opposite
// to Final(), Flip .......
if( ( ((Final.Pos[ Final.E[ fpos ].u ].x - Final.Pos[ F
    ((C[s].Pos[ C[s].E[ spos ].u ].x - C[s].Pos[ C[s].E
    ( ((Final.Pos[ Final.E[ fpos ].u ].x - Final.Pos[ F
    ((C[s].Pos[ C[s].E[ spos ].u ].x - C[s].Pos[ C[s].E
   Flip(s);

// Translate C[s] s.t C[s]{a,b} = F[]{a,b} ........
// Calculate differences between C[s](a) and F[](a) ...
double d_x = rounder( Final.Pos[ Final.E[ fpos ].u ].x
double d_y = rounder( Final.Pos[ Final.E[ fpos ].u ].y

Translate_C(s,d_x,d_y);

// ------------ -------------- ---------- ------- ---
// Place all vertex positions of C[s] into Final[] ..

for(k=0;k<C[s].face_v;k++) {
  Final.Pos[ C[s].F[k] ].x = C[s].Pos[ C[s].F[k] ].x;
  Final.Pos[ C[s].F[k] ].y = C[s].Pos[ C[s].F[k] ].y;
```

```
  }
  for(k=0;k<C[s].inner_v;k++) {
    Final.Pos[ C[s].I[k] ].x = C[s].Pos[ C[s].I[k] ].x;
    Final.Pos[ C[s].I[k] ].y = C[s].Pos[ C[s].I[k] ].y
  }

  // Merge edges of Final[] and C[s] into Final[] ...
  // Check if edge in C[s] is in Final[], if its not Add

  for(k=0;k<C[s].e;k++){       // for each edge in C[s]
     check = 0;
     for(j=0;j<Final.e;j++)     // for each edge in Final
        if( (C[s].E[k].u == Final.E[j].u) &&  (C[s].E[k].
            && (C[s].E[k].z == Final.E[j].z) ){
           check=1;
         }
     if(check==0) {
       // Add edge to Final[] ...
       Final.E[ Final.e ].u = C[s].E[k].u;
       Final.E[ Final.e ].v = C[s].E[k].v;
       Final.E[ Final.e ].z = C[s].E[k].z;
       Final.e++;
     }
   }
  } // end if(not bond) ...

}
```

```
void DFS_T(int v){  // Procedure to merge triconnected em

  int i=0,j=0;

  n = T_number[v] = n + 1;

  for(i=0;i<num_comp;i++)      // for each component i

    if( (Comp_A[v][i].u != 0) || (Comp_A[i][v].u != 0) )

      if(T_number[i]==0){   // i is a new component -> i

        // Merge .......

        Merge_Pos_E(i);
        DFS_T(i);

    }

}

void Merge_3_Block_Tree(int cnt){

  int i=0,j=0,k=0,comp=0;
  double theta=0;

  Pair SAVE[30];
```

```
for(i=0;i<30;i++) { SAVE[i].x = SAVE[i].y = 0.0; }

cout << endl << " ----- --- --- Store C[0] in Final[] -

//  ------------------------------ Store C[0] in Final[
int first = 0, next=0;

// Store all edges in C[0] ...
Final.e = 0;
for(i=0;i<C[ first ].e;i++){
   Final.E[ Final.e ].u = C[ first ].E[i].u;
   Final.E[ Final.e ].v = C[ first ].E[i].v;
   Final.E[ Final.e ].z = C[ first ].E[i].z;
   Final.e++;
}

// Store Inner and Face vertices ...
Final.face_v=0;
for(j=0;j<C[ first ].face_v;j++){
  Final.F[ Final.face_v ] = C[ first ].F[j];
  Final.face_v++;
}

Final.inner_v=0;
for(j=0;j<C[ first ].inner_v;j++){
  Final.I[ Final.inner_v ] = C[ first ].I[j];
  Final.inner_v++;
}
```

```
  // Store all the vertex positions ...
  for(i=0;i<C[ first ].face_v;i++) {
    Final.Pos[ C[ first ].F[i] ].x = C[ first ].Pos[ C[ f
    Final.Pos[ C[ first ].F[i] ].y = C[ first ].Pos[ C[ f
  }
  for(i=0;i<C[ first ].inner_v;i++) {
    Final.Pos[ C[ first ].I[i] ].x = C[ first ].Pos[ C[ f
    Final.Pos[ C[ first ].I[i] ].y = C[ first ].Pos[ C[ f
  }

  n=0;
  DFS_T(first);

  cout << " -------- -- FINAL COORDINATES ----- ---- " <<
  for(j=1;j<nodes+1;j++){
     cout << "[" << j << "](" << Final.Pos[ j ].x << ","
               << Final.Pos[ j ].y << ")" << endl;
  }
  cout << endl << endl;

  getchar();
}


void Mult_Edge(int t, int u, int v, int num){

   // Input: Component #, Edge(u,v), Number of edges to a
```

```
//  ... m_edge starts at |V| + 1 ..


for(int i=0;i<num;i++) {       // for each pair of vir

  // Multilink (a,b) .. Create virtual edges (u,m_edge

  cout << "New Vertex = " << m_edge << endl;
  C[t].E[ C[t].e ].u = u; C[t].E[ C[t].e ].v = m_edge;
  C[t].e++;

  C[t].E[ C[t].e ].u = m_edge; C[t].E[ C[t].e ].v = v;
  C[t].e++;
  // Mark vertex m_edge as special ...
  C[t].V[m_edge] = 1;

  /* Draw virtual edges above ...
  cout << "ABOVE" << endl;
  C[t].Pos[m_edge].x = (C[t].Pos[u].x + C[t].Pos[v].x)
  C[t].Pos[m_edge].y = C[t].Pos[u].x + 1.0;
  */

  // Draw virtual edges below ...
  cout << "BELOW" << endl;
  C[t].Pos[m_edge].x = (C[t].Pos[u].x + C[t].Pos[v].x)
  C[t].Pos[m_edge].y = C[t].Pos[u].x - (1.0 + num );

  // Print out new vertex positions ...
```

```
      cout << "Pos[" << m_edge << "]={" << C[t].Pos[m_edge
           << "," << C[t].Pos[m_edge].y << ")" << endl;


    m_edge++;

   }
}

double Get_Angle_X_Axis_F(int a, int b){

    // Calculate angle from pos. x-axis to b ..
    // Distances are created from b(x,y) - a(x,y) ...

    double dx = Final.Pos[b].x - Final.Pos[a].x; if(fabs(
    double dy = Final.Pos[b].y - Final.Pos[a].y; if(fabs(
    double theta = 0;

    // cout << "{dx,dy} = {" << dx << "," << dy << "}" <<

    if((dx>0)&&(dy==0))     // In position, don't rotate
        theta = 0;

    if((dx != 0)&&(dy > 0)){          // Quadrant I & Qua
       theta = (180.0/double(PI))*acos(dx/sqrt(pow(dx,2)+
    }
    else if((dx != 0)&&(dy < 0)){     // Quadrant III & Q
       theta = 360 - (180.0/double(PI))*acos(dx/sqrt(pow(
```

```
    }
    else if((dx==0)&&(dy > 0)){  // Vertical Line b_y > a
       theta = 90;
    }
    else if((dx==0)&&(dy < 0)){  // Vertical Line a_y > b
       theta = 270;
    }
    else if((dx < 0)&&(dy==0)){  // Horizontal Line b_x <
       theta = 180;
    }

    // cout << "theta = " << theta << endl;
    return(theta);

}

void Translate_F(double shiftx,double shifty){
  // Translate Final[] shiftx in x-direction and shifty i

  // cout << "SHIFT (" << shiftx << "," << shifty << ")"

  for(j=0;j<Final.face_v;j++){
    Final.Pos[ Final.F[j] ].x = Final.Pos[ Final.F[j] ].x
    Final.Pos[ Final.F[j] ].y = Final.Pos[ Final.F[j] ].y
  }
  for(j=0;j<Final.inner_v;j++){
    Final.Pos[ Final.I[j] ].x = Final.Pos[ Final.I[j] ].x
    Final.Pos[ Final.I[j] ].y = Final.Pos[ Final.I[j] ].y
```

```
  }
}

void Rotate_F(double theta){      // Rotate Final[] theta

  Pair Temp[30];         // Temporary storage ...
  for(int j=0;j<30;j++) Temp[j].x = Temp[j].y = 0.0;

  for(j=1;j<nodes+1;j++){

    // Change angle in degrees to radians ..
    Temp[j].x = Final.Pos[ j ].x*cos(theta*PI/180) + Fina
    Temp[j].y = -Final.Pos[ j ].x*sin(theta*PI/180) + Fin

    // Round coordinates to nearest integer ..
    if(fabs(Temp[j].x) < TOL) Temp[j].x = 0;
    if(fabs(Temp[j].y) < TOL) Temp[j].y = 0;
    if(fabs(Temp[j].x - ceil(Temp[j].x)) < TOL2 ) Temp[j]
    if(fabs(Temp[j].y - ceil(Temp[j].y)) < TOL2 ) Temp[j]
  }

  // Store in Coordinate lists ...
  for(j=1;j<nodes+1;j++){
    Final.Pos[ j ].x = Temp[j].x;
    Final.Pos[ j ].y = Temp[j].y;
  }

}
```

```
double Get_Angle_X_Axis_C(int t,int a, int b){

    // Calculate angle from pos. x-axis to b ..
    // Distances are created from b(x,y) - a(x,y) ...

    double dx = C[t].Pos[b].x - C[t].Pos[a].x; if(fabs(dx
    double dy = C[t].Pos[b].y - C[t].Pos[a].y; if(fabs(dy
    double theta = 0;

    if((dx>0)&&(dy==0))     // In position, don't rotate
        theta = 0;

    if((dx != 0)&&(dy > 0)){          // Quadrant I & Qua
       theta = (180.0/double(PI))*acos(dx/sqrt(pow(dx,2)+
    }
    else if((dx != 0)&&(dy < 0)){     // Quadrant III & Q
       theta = 360 - (180.0/double(PI))*acos(dx/sqrt(pow(
    }
    else if((dx==0)&&(dy > 0)){  // Vertical Line b_y > a
       theta = 90;
    }
    else if((dx==0)&&(dy < 0)){  // Vertical Line a_y > b
       theta = 270;
    }
    else if((dx < 0)&&(dy==0)){  // Horizontal Line b_x <
       theta = 180;
    }
```

```
    // cout << "theta = " << theta << endl;
    return(theta);

}

void Translate_C(int t, double shiftx,double shifty){
  // Translate component t shiftx in x-direction and shif

  for(j=0;j<C[t].face_v;j++){
    C[t].Pos[ C[t].F[j] ].x = C[t].Pos[ C[t].F[j] ].x + s
    C[t].Pos[ C[t].F[j] ].y = C[t].Pos[ C[t].F[j] ].y + s

    // Round coordinates to nearest integer ..
    if(fabs( C[t].Pos[C[t].F[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].F[j] ].x - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].x = ceil(C[t].Pos[C[t].F[j]].x)
    if(fabs( C[t].Pos[C[t].F[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].F[j]].y = ceil(C[t].Pos[C[t].F[j]].y)
  }
  for(j=0;j<C[t].inner_v;j++){
    C[t].Pos[ C[t].I[j] ].x = C[t].Pos[ C[t].I[j] ].x + s
    C[t].Pos[ C[t].I[j] ].y = C[t].Pos[ C[t].I[j] ].y + s

    if(fabs( C[t].Pos[C[t].I[j] ].x ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].y ) < TOL) C[t].Pos[C[t
    if(fabs( C[t].Pos[C[t].I[j] ].x - ceil(C[t].Pos[C[t].
```

```
      C[t].Pos[C[t].I[j]].x = ceil(C[t].Pos[C[t].I[j]].x)
    if(fabs( C[t].Pos[C[t].I[j] ].y - ceil(C[t].Pos[C[t].
      C[t].Pos[C[t].I[j]].y = ceil(C[t].Pos[C[t].I[j]].y)
  }
}

void Rotate_C(int t, double theta){   // Rotate component

  Pair Temp[30];        // Temporary storage ...
  for(int j=0;j<30;j++) Temp[j].x = Temp[j].y = 0.0;

  for(j=0;j<C[t].face_v;j++){

    // Change angle in degrees to radians ..
    Temp[j].x = C[t].Pos[ C[t].F[j] ].x*cos(theta*PI/180)
      + C[t].Pos[ C[t].F[j] ].y*sin(theta*PI/180);
    Temp[j].y = -C[t].Pos[ C[t].F[j] ].x*sin(theta*PI/180
     + C[t].Pos[ C[t].F[j] ].y*cos(theta*PI/180);

    // Round coordinates to nearest integer ..
    if(fabs(Temp[j].x) < TOL) Temp[j].x = 0;
    if(fabs(Temp[j].y) < TOL) Temp[j].y = 0;
    if(fabs(Temp[j].x - ceil(Temp[j].x)) < TOL2 )
        Temp[j].x = ceil(Temp[j].x);
    if(fabs(Temp[j].y - ceil(Temp[j].y)) < TOL2 )
       Temp[j].y = ceil(Temp[j].y);
  }
  // Store in Coordinate lists ...
```

```
  for(j=0;j<C[t].face_v;j++){
    C[t].Pos[ C[t].F[j] ].x = Temp[j].x;
    C[t].Pos[ C[t].F[j] ].y = Temp[j].y;
  }

  for(j=0;j<C[t].inner_v;j++){

    // Change angle in degrees to radians ..
    Temp[j].x = C[t].Pos[ C[t].I[j] ].x*cos(theta*PI/180)
        + C[t].Pos[ C[t].I[j] ].y*sin(theta*PI/180);
    Temp[j].y = -C[t].Pos[ C[t].I[j] ].x*sin(theta*PI/180
       + C[t].Pos[ C[t].I[j] ].y*cos(theta*PI/180);

    // Round coordinates to nearest integer ..
    if(fabs(Temp[j].x) < TOL) Temp[j].x = 0;
    if(fabs(Temp[j].y) < TOL) Temp[j].y = 0;
    if(fabs(Temp[j].x - ceil(Temp[j].x)) < TOL2 ) Temp[j]
    if(fabs(Temp[j].y - ceil(Temp[j].y)) < TOL2 ) Temp[j]
  }

  // Store in Coordinate lists ...
  for(j=0;j<C[t].face_v;j++){
    C[t].Pos[ C[t].I[j] ].x = Temp[j].x;
    C[t].Pos[ C[t].I[j] ].y = Temp[j].y;
  }

}
```

```
double rounder(double r){
   if( (fabs(r) - 0.0) < .001 )
      r = 0.0;
   return(r);
}

double cool(int i){
   double val=0;
   val = ( sqrt(nodes/PI) / ( 1 + (PI/nodes)*pow(i,1.5) )
   return(val);
}

void Draw_T(int t){   // Draw triconnected comp i s.t. vi

   int i=0,j=0,cnt=0,min_v=0,nextv=0,tmp_st=0,tmp_end=0;
   int st=0,end=0,newst=0,vi=0,stop=0,mn=0,mx=0;

   // cout << endl << "Drawing Component #" << t << " |E|

   // getchar();

   // Reorder s.t u < v for all (u,v) ..
   for(i=0;i<C[t].e;i++) {
     mn = min(C[t].E[i].u,C[t].E[i].v);
     mx = max(C[t].E[i].u,C[t].E[i].v);
     C[t].E[i].u = mn;
     C[t].E[i].v = mx;
   }
```

```
// Print edges
for(i=0;i<C[t].e;i++)
  cout << "(" << C[t].E[i].u << "," << C[t].E[i].v <<
cout << endl;

// getchar();

if( (strcmp(Type[C[t].type],"POLYGONS")==0) ){

   cout << " ------- POLYGON ----------- " << endl;

   st = C[t].F[0] = C[t].E[0].u;       // Store vertic
   end = C[t].F[1] = C[t].E[0].v;

   // cout << "Face = {" << st << "," << end << ", ...
   // getchar();

   stop==0; cnt=2;
   while(stop == 0){     // Loop until entire face is

     int stop2=0; i=0;

     while((stop2==0) && (i<C[t].e) ){           // Se

        if( (C[t].E[i].u == end) && (C[t].E[i].v != st
            // cout << "Start 1 .. st= " << st << " en
            // cout << "1 .. (" << C[t].E[i].u << ","
```

```
            C[t].F[cnt] = C[t].E[i].v;
            tmp_st = end; tmp_end = C[t].E[i].v;
            // cout << "End 1 .. st= " << tmp_st << "
            stop2 = 1;
        }
        else if( (C[t].E[i].v == end) && (C[t].E[i].u
            // cout << "Start 2 .. st= " << st << " en
            // cout << "2 .. (" << C[t].E[i].u << ","
            C[t].F[cnt] = C[t].E[i].u;
            tmp_st = end; tmp_end = C[t].E[i].u;
            // cout << "End 2 .. st= " << tmp_st << "
            stop2 = 1;
        }
        i++;
    }

    st = tmp_st; end = tmp_end;     // Save new start

     // cout << endl << "Count = " << cnt << " .....
     //    << C[t].F[cnt] << endl;
     // getchar();

    if(cnt == (C[t].e - 1) ) {
        stop = 1;
        C[t].face_v = cnt + 1;
        // cout << "C[" << t << "].face_v = " << C[t].
    }
```

```
      cnt++;

   } // end of while()

}
else if( (strcmp(Type[C[t].type],"BONDS")==0) ){

   cout << " ------ BOND -------- " << endl;

   int virt = 0;                  // Count the number of
   for(i=0;i<C[t].e;i++)
     if(C[t].E[i].z != 0) virt++;

   // Position vertices (u,v) ....
   C[t].Pos[ C[t].E[0].u ].x = 0.0;
   C[t].Pos[ C[t].E[0].u ].y = 0.0;

   C[t].Pos[ C[t].E[0].v ].x = radius;
   C[t].Pos[ C[t].E[0].v ].y = 0.0;

   // Print out new vertex positions ...
   cout << "Pos[" << C[t].E[0].u << "]={" << C[t].Pos[
        << "," << C[t].Pos[ C[t].E[0].u ].y << ")" <<
   cout << "Pos[" << C[t].E[0].v << "]={" << C[t].Pos[
        << "," << C[t].Pos[ C[t].E[0].v ].y << ")" <<


   if( (C[t].e - virt - 1) > 0){
```

```
      // Mult_Edge(t,C[t].E[0].u,C[t].E[0].v,C[t].e -
      C[t].d_bond = 1;
   }

}
else if( (strcmp(Type[C[t].type],"TRI")==0) ){

  cout << " ------- TRICONNECTED G ------ " << endl;

  // Find the virtual edge ...

  for(i=0;i<C[t].e;i++)
     if (C[t].E[i].z != 0){

       vi = i;
       // cout << "Virtual Edge (" << C[t].E[i].u << "
       //  << "," << C[t].E[i].z << ")" << endl;

       st = min(C[t].E[i].u,C[t].E[i].v);          // s
       // cout << "Start= " << min(C[t].E[i].u,C[t].E[

       end = max(C[t].E[i].u,C[t].E[i].v);         // e
       // cout << "End= " << max(C[t].E[i].u,C[t].E[i]
     }

   // getchar();

   cnt = 0;
```

```
C[t].F[cnt] = st;

cnt++;

min_v = 100;
for(i=0;i<C[t].e;i++)                  // for each edge
  if(C[t].E[i].u == st){               // starting at s
      if( C[t].E[i].v < min_v ) {
         nextv =  C[t].E[i].v;
         min_v = C[t].E[i].v;
      }
   }

C[t].F[cnt] = min_v;
cnt++;

newst = min_v;

// Generate outer face F of C[] ...

stop = 0;                             // Stopping condi
while( stop == 0 ) {

  min_v = 100;

  for(i=0;i<C[t].e;i++)                  // for each ed
    if(C[t].E[i].u == newst){            // starting at
```

```
    if( C[t].E[i].v < min_v ) {
       nextv =  C[t].E[i].v;
       min_v = C[t].E[i].v;
    }

    // check for end ...
    if(C[t].E[i].v == end) {
       nextv =  C[t].E[i].v;
    }

  }

// Store outer face ...
C[t].F[cnt] = nextv;
cnt++;

// getchar();

// Case 1  .............  nextv = end ...
if(nextv == end){
    // cout << "CASE 1 ---- STOP ---- " << endl;
    C[t].face_v = cnt;
    stop = 1;
}
else {   // Case 2 ... (nextv - -> st) && (next,e

  // Check if (next,end) in F[]
  // Check (F[0],F[1]), (F[1],F[2]), ... , (F[cnt
```

```
        for(i=0;i<(cnt-1);i++)
           if( (C[t].F[i]==st) && (C[t].F[i+1]==end) )
             // Check if (nextv - ->  st)
             for(i=0;i<C[t].e;i++)              // for

               if( (C[t].E[i].u == st) && (C[t].E[i].v
                   // cout << "CASE 2 ---- STOP ---- "
                   C[t].face_v = cnt;
                   stop = 1;
               }
     }

     newst = nextv;

  }  // end while()

}  // end if(TRI)


if( (strcmp(Type[C[t].type],"BONDS")!= 0) ){

 // Continue ................. Print out outer face o

 //  ---------------------------------------------
 int STR[30],chck=0;
 for(i=0;i<30;i++) STR[i]=0;
```

```
STR[0] = C[t].E[0].u; cnt=1;

for(i=1;i<C[t].e;i++){           // Search E[].u
   chck = 0;
   for(j=0;j<cnt;j++)
      if( C[t].E[i].u == STR[j] ) chck = 1;
   if(chck == 0) { STR[cnt] = C[t].E[i].u; cnt++; }
}

for(i=0;i<C[t].e;i++){           // Search E[].v
   chck = 0;
   for(j=0;j<cnt;j++)
      if(C[t].E[i].v == STR[j]) chck = 1;
   if(chck == 0) { STR[cnt] = C[t].E[i].v; cnt++; }
}

// Find and Store Inner vertices ...
C[t].inner_v = 0;

for(i=0;i<cnt;i++){
   chck = 0;
   for(j=0;j<C[t].face_v;j++)
       if(STR[i]==C[t].F[j]) chck=1;
   if(chck==0) { C[t].I[ C[t].inner_v ] = STR[i]; C[t
}

// Initialize force and position data structures ...
for(i=0;i<15;i++){
```

```
   Fr[i].x = Fr[i].y = 0;
   Fa.x = Fa.y = 0;
   C[t].Pos[i].x = C[t].Pos[i].y = 0.0;
 }

 // cout << endl << "---- -------- -- START ---- -----

// Position all vertices of an outer face W in vertice
// of a regular polygon of size k inscribed into the u
// and place all other vertices in the origin ...

double angle = (double(2*PI)/C[t].face_v);
// Central angle between each outer face vertices

cnt=0;
for(i=0;i<C[t].face_v;i++){      // Position outer fac
   C[t].Pos[ C[t].F[i] ].x = rounder( radius * cos(ang
   C[t].Pos[ C[t].F[i] ].y = rounder( radius * sin(ang
   cnt++;
}

for(i=0;i<C[t].inner_v;i++)     // Position inner vert
  C[t].Pos[ C[t].I[i] ].x = C[t].Pos[ C[t].I[i] ].y =

for(i=0;i<iterations;i++){        // Main Loop -------

   // For all vertices v in V, set resultant forces Fr
```

```
for(j=0;j<C[t].face_v;j++)  Fr[ C[t].F[j] ].x = Fr[
for(j=0;j<C[t].inner_v;j++) Fr[ C[t].I[j] ].x = Fr[

// For all edges (u,v) in E, calculate attractivr f
// update Fr[u] and Fr[v]

for(j=0;j<C[t].e;j++){

   int u = C[t].E[j].u; int v = C[t].E[j].v;

   double distx = (C[t].Pos[u].x - C[t].Pos[v].x);
   double disty = (C[t].Pos[u].y - C[t].Pos[v].y);

   Fa.x = (sqrt(nodes/PI))*(pow(distx,3));
   Fa.y = (sqrt(nodes/PI))*(pow(disty,3));


   if(distx > 0){   // U_x > V_x
      Fr[u].x = Fr[u].x - fabs(Fa.x);  // Move left
      Fr[v].x = Fr[v].x + fabs(Fa.x);  // Move righ
   }else {          // U_x < V_x .....
      Fr[u].x = Fr[u].x + fabs(Fa.x);  // Move righ
      Fr[v].x = Fr[v].x - fabs(Fa.x);  // Move left
   }

   if(disty > 0){   // U_x > V_x
      // cout << u << " .. y change = " << (-1) * a
      Fr[u].y = Fr[u].y - fabs(Fa.y);  // Move down
```

```
      // cout << v << " .. y change = " << abs(Fa.y
      Fr[v].y = Fr[v].y + fabs(Fa.y);  // Move up .
   }else {          // U_x < V_x .....
      // cout << u << " .. y change = " << abs(Fa.y
      Fr[u].y = Fr[u].y + fabs(Fa.y);  // Move up .
      // cout << v << " .. y change = " << (-1) * a
      Fr[v].y = Fr[v].y - fabs(Fa.y);  // Move down
   }

}

// For all vertices v in (V - W) {INNER VERTICES},
// move vertex v in direction of force ..

for(j=0;j<C[t].inner_v;j++) {

     if(Fr[ C[t].I[j] ].x != 0){
         C[t].Pos[ C[t].I[j] ].x = C[t].Pos[ C[t].I
            + min(fabs(Fr[ C[t].I[j] ].x),cool(i))
            * (Fr[ C[t].I[j] ].x/fabs(Fr[ C[t].I[j]
     }

     if(Fr[ C[t].I[j] ].y != 0){
         C[t].Pos[ C[t].I[j] ].y = C[t].Pos[ C[t].I
            + min(fabs(Fr[ C[t].I[j] ].y),cool(i))
            * (Fr[ C[t].I[j] ].y/fabs(Fr[ C[t].I[j]
     }
}
```

```
   }  // end of main loop.

   // Print out coordinates ...

   cout << " --- " << t << " -------- AFTER " << i << " I
   for(j=0;j<C[t].face_v;j++)
     cout << "[" << C[t].F[j] << "]: (" << C[t].Pos[ C[t]
                                  << "," << C[t].Pos[ C[t]
   for(j=0;j<C[t].inner_v;j++)
     cout << "[" << C[t].I[j] << "]: (" << C[t].Pos[ C[t]
                                  << "," << C[t].Pos[ C[t]

   getch();


   }  // end if( not bond )

}

int PATHCHECK(int v, int x){
// Check if a path of tree edges connects v and x

  int
  i,node;

  if(v < x)
    if(Edge[v][x]==1)
```

```
      global_check = 1;
    else {
      for(i=0;i<A_1[v].deg;i++){
node = A_1[v].neigh[i];
if(Edge[v][node]==1)
  PATHCHECK(node,x);
      }
    }

  return(global_check);
}

void DELETE_E(int u, int v, int z){ // Delete e from ESTA

  // cout << "---> Delete (" << u << "," << v << "," << z

  if ( (ESTACK[estack-1].u == u) && (ESTACK[estack-1].v =
       && (ESTACK[estack-1].z == z) ) {
    ESTACK[estack-1].u = 0;
    ESTACK[estack-1].v = 0;
    ESTACK[estack-1].z = 0;
  }
  else {
    cout << "Problem in DELETE ESTACK (Terminate Program)
    exit(1);
  }

  estack--;
```

```
}

void ADD_E(int u, int v, int z){   // Add e to ESTACK

  // cout << "---> Add (" << u << "," << v << "," << z <<

  ESTACK[estack].u = min(u,v);
  ESTACK[estack].v = max(u,v);
  ESTACK[estack].z = z;

  estack++;

}

void DELETEA(int h, int a, int b){     // Delete (h,a,b)

  Last_Deleted.h = h;
  Last_Deleted.a = a;
  Last_Deleted.b = b;

  if((TSTACK[tstack-1].h == h)
     && (TSTACK[tstack-1].a == a)
     && (TSTACK[tstack-1].b == b)) {

    TSTACK[tstack-1].h = 0;
    TSTACK[tstack-1].a = 0;
    TSTACK[tstack-1].b = 0;
```

```
  }
  else {
    cout << "Problem in DELETE TSTACK (Terminate Program)
    getchar();
    exit(1);
  }

  tstack--;

}

void ADD(int h, int a, int b){   // Add (h,a,b) to TSTACK

  // cout << "---> ADD (" << h << "," << a << "," << b <<
  // getchar();

  if(a < b){
    TSTACK[tstack].h = h;
    TSTACK[tstack].a = a;
    TSTACK[tstack].b = b;
  }
  else {
    TSTACK[tstack].h = h;
    TSTACK[tstack].a = b;
    TSTACK[tstack].b = a;
  }

  tstack++;
```

```
}

void PATHSEARCH(int v){    // Find split components of G

  int option,i,k,y,l,p,q,g,check,del_check,saveda=0,saved
  int h,a,b,w,x,z1,z2,v1,v2,w1,w2,z,e1,e2,saved_x=0,saved
  int num_edge=0;

  for(i=0;i<A_1[v].deg;i++){     // for each w in A[v]

    w = A_1[v].neigh[i];

    if(Edge[v][w]==1){             // if v -> w

      check = 0;
      del_check = 0;

      for(l=0;l<path;l++)      // for each path ...

if((v==Paths[l].P[0].u)&&(w==Paths[l].P[0].v)){
        // if (v,w) is the first edge of a path ..

  y = 0;

  if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
```

```
  curr_b = TSTACK[tstack-1].b;
  }
else curr_h = curr_a = curr_b = 0;

// while (h,a,b) on TSTACK has a > LOWPT1[w] ..
while(curr_a > LOWPT1[w]){

  y = max(y,curr_h);

  DELETEA(curr_h,curr_a,curr_b);          // DELETE (h,a

  del_check = 1;

  if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
    curr_b = TSTACK[tstack-1].b;
  }
  else curr_h = curr_a = curr_b = 0;

}

// if no triples were deleted .. add ( w+ND[w]-1 , LOWP
if(del_check==0)  ADD( w+ND[w]-1 , LOWPT1[w] , v );

// else if (h,a,b) last triple deleted ..
        // add ( max(y,w+ND[w]-1) , LOWPT1[w] , b ) to
else ADD(max(y,w+ND[w]-1),LOWPT1[w],Last_Deleted.b);
```

```
  ADD(0,0,0);  // Add end of stack marker ..

}

      PATHSEARCH(w);

      r = MARK[v][w];
      ADD_E(v,w,r);   // ADD (v,w) to ESTACK

      // TEST FOR TYPE II ..........

      if(tstack>0){
        curr_h = TSTACK[tstack-1].h;
        curr_a = TSTACK[tstack-1].a;
        curr_b = TSTACK[tstack-1].b;
      }
      else curr_h = curr_a = curr_b = 0;


      while( (v!=1) && (   ((DEGREE[w]==2) && (A1[w] > w)

if ( (curr_a == v) && (FATHER[curr_b]==curr_a) ){

  DELETEA(curr_h,curr_a,curr_b);    // DELETE (h,a,b) fro

}
```

```
else {


  if( (DEGREE[w]==2) && (A1[w]>w) ){

    j++;

            SepPair[seppair].u = v; SepPair[seppair].v =
            seppair++;


            // Add top two edges (v,w) and (w,x) on ESTAC

            // cout << "num_comp = " << num_comp << endl;
            // cout << "--- (type 2) --- NEW COMPONENT --

            num_edge = 0;

    // cout << "(" << ESTACK[estack-1].u << "," << ESTACK
    //  <<  ESTACK[estack-1].z << ")" << endl;

    int tu = ESTACK[estack-1].u;
    int tv = ESTACK[estack-1].v;
    int tz = ESTACK[estack-1].z;

    C[num_comp].E[num_edge].u = tu;
    C[num_comp].E[num_edge].v = tv;
```

```
C[num_comp].E[num_edge].z = tz;

num_edge++;

        DELETE_E(tu,tv,tz);

DEGREE[tu]--;
DEGREE[tv]--;

        // cout << "(" << ESTACK[estack-1].u << "," <
//   <<  ESTACK[estack-1].z << ")" << endl;

tu = ESTACK[estack-1].u;
tv = ESTACK[estack-1].v;
tz = ESTACK[estack-1].z;

C[num_comp].E[num_edge].u = tu;
C[num_comp].E[num_edge].v = tv;
C[num_comp].E[num_edge].z = tz;

num_edge++;

x = ESTACK[estack-1].v;

DELETE_E(tu,tv,tz);

DEGREE[tu]--;
DEGREE[tv]--;
```

```
        // cout << "(" << v << "," << x << "," << j <

C[num_comp].E[num_edge].u = v;
C[num_comp].E[num_edge].v = x;
C[num_comp].E[num_edge].z = j;

        C[num_comp].type = POLYGONS;    // 3 - Polygo

        num_edge++;
        C[num_comp].e = num_edge;
        num_comp++;

// if( (y,z) on ESTACK has (y,z)=(x,v) )

if(estack>0){
  curr_x = ESTACK[estack-1].u;
  curr_y = ESTACK[estack-1].v;
  curr_z = ESTACK[estack-1].z;
} else curr_x = curr_y = curr_z = 0;


if ( ((curr_x == x) && (curr_y == v)) || ((curr_x ==

  FLAG1 = TRUE;
  saved_x = curr_x;
  saved_y = curr_y;
  DELETE_E(curr_x,curr_y,curr_z);     // DELETE (y,z)
```

```
  }

}  // end if()

else if( (curr_a == v) && ( curr_a != FATHER[curr_b] ))

          SepPair[seppair].u = v; SepPair[seppair].v =

  j=j+1;

  DELETEA(curr_h,curr_a,curr_b);     // DELETE (h,a,b)

          // while ( ((x,y) on ESTACK has (a <= x <= h)

  if(estack>0){
    curr_x = ESTACK[estack-1].u;
    curr_y = ESTACK[estack-1].v;
    curr_z = ESTACK[estack-1].z;
  } else curr_x = curr_y = curr_z = 0;

          // cout << "num_comp = " << num_comp << endl;
          // cout << "(type 2) --- NEW COMPONENT ------

          num_edge = 0;

          // getchar();
```

```
    while ( (curr_a <= curr_x) && (curr_x <= curr_h)
                && (curr_a <= curr_y) && (curr_y <= curr_h

                if(estack>0){
curr_x = ESTACK[estack-1].u;
curr_y = ESTACK[estack-1].v;
curr_z = ESTACK[estack-1].z;
                } else curr_x = curr_y = curr_z = 0;


       if((curr_x==curr_a) && (curr_y==curr_b)) {  // if

  FLAG1 = TRUE;    // Split component is an edge ..

  saved_x = curr_x;  saved_y = curr_y;
  saveda  = curr_a;  savedb = curr_b;

  DELETE_E(curr_x,curr_y,curr_z);
                  // Delete (a,b) from ESTACK and save ..

          if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
    curr_b = TSTACK[tstack-1].b;
                  }
  else curr_h = curr_a = curr_b = 0;

       }
```

```
                else {

 x = curr_x; y = curr_y; z = curr_z;

 // Add (x,y) to current component ...

 C[num_comp].E[num_edge].u = x;
 C[num_comp].E[num_edge].v = y;
 C[num_comp].E[num_edge].z = z;

 num_edge++;

 DELETE_E(curr_x,curr_y,curr_z);
                // Delete (x,y) from ESTACK

 DEGREE[x]--;
 DEGREE[y]--;

              }

              if(estack>0){
curr_x = ESTACK[estack-1].u;
curr_y = ESTACK[estack-1].v;
curr_z = ESTACK[estack-1].z;
      } else curr_x = curr_y = curr_z = 0;

          }  // end while()
```

```
// Add saved edge (a,b,j) to new component ..
        // cout << "(" << saveda << "," << savedb <<

        C[num_comp].E[num_edge].u = saveda;
C[num_comp].E[num_edge].v = savedb;
C[num_comp].E[num_edge].z = j;
num_edge++;

C[num_comp].e = num_edge;

        // Check the type of this new split component
        // Check degree of first vertex u found in th
        v_cnt = 0; v_one = 0; c = 0;
        for(c=0;c<C[num_comp].e;c++)  {

          if(c==0){ v_one = C[num_comp].E[0].u; v_cnt
          else {
            if( (C[num_comp].E[c].u == v_one) || (C[n
             v_cnt++;
            }

        }

        // cout << "1st vertex " << v_one << " .. cou
        if(v_cnt > 2) C[num_comp].type = TRI;    // T
        else C[num_comp].type = POLYGONS;    // Trico
        // cout << "Type " << Type[C[num_comp].type]
```

```
            num_comp++;

  x = savedb;

}  // end else if()

if(FLAG1 == TRUE){       // split component is an edge

  FLAG1 = FALSE; j++;

            // cout << "num_comp = " << num_comp << endl;
            // cout << endl << "(FLAG1 = TRUE) ----- NEW

  // Add saved edge, (x,v,j-1) and (x,v,j) to new compo
   // cout << "(" << saved_x << "," << saved_y << ",0)"
    //  << "(" << saved_x << "," << saved_y << "," << j
    //  << "(" << saved_x << "," << saved_y << "," << j

  C[num_comp].E[0].u = saved_x;
  C[num_comp].E[0].v = saved_y;
  C[num_comp].E[0].z = j-1;
  C[num_comp].E[1].u = saved_x;
  C[num_comp].E[1].v = saved_y;
  C[num_comp].E[1].z = 0;
  C[num_comp].E[2].u = saved_x;
  C[num_comp].E[2].v = saved_y;
  C[num_comp].E[2].z = j;
```

```
      C[num_comp].e = 3;
      C[num_comp].type = BONDS;    // Multilink ...
      num_comp++;

              // cout << "num_comp = " << num_comp << endl;

              getchar();

      DEGREE[saved_x]--; DEGREE[saved_y]--;

    }

    // Add (v,x,j) to ESTACK
    ADD_E(v,x,j);

            DEGREE[x]++;
    DEGREE[v]++;

    FATHER[x] = v;

    global_check = 0;
    if( PATHCHECK(A1[v],x)==1 )   // if A1[v] -> * x
      A1[v] = x;

    w = x;

  }  // end else()
```

```
if(tstack>0){
  curr_h = TSTACK[tstack-1].h;
  curr_a = TSTACK[tstack-1].a;
  curr_b = TSTACK[tstack-1].b;
}
else curr_h = curr_a = curr_b = 0;

      }  // end while( type 2 pair loop )

      if(estack>0){
  curr_x = ESTACK[estack-1].u;
  curr_y = ESTACK[estack-1].v;
  curr_z = ESTACK[estack-1].z;
} else curr_x = curr_y = curr_z = 0;

      // Check that there are more than three vertices re
      // where all three have degree=2.

      if( ( DEGREE[LOWPT1[w]] >2) &&
        (LOWPT2[w] >= v) && ( (LOWPT1[w] != 1)
           || (FATHER[v] != 1) || (w > 3) ) ){

        SepPair[seppair].u = LOWPT1[w]; SepPair[seppair].

j++;

        // cout << "------- NEW COMPONENT -- C -------- "
```

```
        // while (x,y) on estack has ...
while( ((w <= curr_x) && (curr_x < (w + ND[w]))) || ((w <
          && (curr_y < (w + ND[w]))) ){

        // begin ...
  x = curr_x; y = curr_y; z = curr_z;

  // Add (x,y) to new component ..
  // cout << "(" << x << "," << y << "," << z << ") ... "

        C[num_comp].E[num_edge].u = x;
  C[num_comp].E[num_edge].v = y;
  C[num_comp].E[num_edge].z = z;

  num_edge++;

        // delete (x,y) from estack ..
  DELETE_E(x,y,z);

        // decrement degree(x), degree(y)
  DEGREE[x]--;
  DEGREE[y]--;

  if(estack>0){
    curr_x = ESTACK[estack-1].u;
    curr_y = ESTACK[estack-1].v;
    curr_z = ESTACK[estack-1].z;
```

```
  } else curr_x = curr_y = curr_z = 0;

}

// Add (v,LOWPT1[w],j) to new component ..
        // cout << "(" << v << "," << LOWPT1[w] << "," <<

C[num_comp].E[num_edge].u = v;
C[num_comp].E[num_edge].v = LOWPT1[w];
C[num_comp].E[num_edge].z = j;

        num_edge++;

C[num_comp].e = num_edge;

        // Check the type of this new split component ..
        // Check degree of first vertex u found in the fi
        v_cnt = 0; v_one = 0; c = 0;
        for(c=0;c<C[num_comp].e;c++)  {

              if(c==0){ v_one = C[num_comp].E[0].u; v_cnt
              else {
                if( (C[num_comp].E[c].u == v_one) ||
                   (C[num_comp].E[c].v == v_one) )
                   v_cnt++;
                }

            }
```

```
        // cout << "1st vertex " << v_one << " .. count=

        if(v_cnt > 2) C[num_comp].type = TRI;    // Tri.
        else C[num_comp].type = POLYGONS;    // Triconnec

        num_comp++;

if (A1[v] == w) A1[v] = LOWPT1[w];

        // Test for multiple edges ...

if(estack>0){
  curr_x = ESTACK[estack-1].u;
          curr_y = ESTACK[estack-1].v;
          curr_z = ESTACK[estack-1].z;
}

if ( ((curr_x==v)&&(curr_y==LOWPT1[w]))
             || ((curr_x==LOWPT1[w]) && (curr_y==v)) ) {

  j = j+1;

  cout << "MULTIPLE EDGE SCENARIO" << endl;

  // Add (x,y) (v,LOWPT1[w],j-1) (v,LOWPT1[w],j) to new c

  /* cout << " ---------------- NEW COMP.----------------
```

```
  cout << "(" << curr_x << "," << curr_y << ")" << endl;
  cout << "(" << v << "," << LOWPT1[w] << "," << j-1 << "
  cout << "(" << v << "," << LOWPT1[w] << "," << j << ")"
          */

  DEGREE[v]--;
  DEGREE[LOWPT1[w]]--;
}

if(LOWPT1[w] != FATHER[v]){

  // ( v, Lowpt1(w) ) is a separation pair ...
  // Check if it is also an edge ...

          if(Edge[v][LOWPT1[w]] != 0){ //   ( v, Lowpt1(w

  j++;

          /* cout << "num_comp = " << num_comp << endl;
  cout << "----- NEW SPLIT COMPONENT ------------------
  cout << "(" << v << "," << LOWPT1[w] << "," << j-1 << "
  cout << "(" << v << "," << LOWPT1[w] << "," << 0 << ")
  cout << "(" << v << "," << LOWPT1[w] << "," << j << ")"
          */

          MARK[v][LOWPT1[w]] = MARK[LOWPT1[w]][v] = j;
          // cout << "Mark [" << v << "][" << LOWPT1[w] <
```

```
C[num_comp].E[0].u = v;
C[num_comp].E[0].v = LOWPT1[w];
        C[num_comp].E[0].z = j-1;
C[num_comp].E[1].u = v;
C[num_comp].E[1].v = LOWPT1[w];;
        C[num_comp].E[1].z = 0;
C[num_comp].E[2].u = v;
C[num_comp].E[2].v = LOWPT1[w];;
        C[num_comp].E[2].z = j;
C[num_comp].type = BONDS;   // Multilink ..
C[num_comp].e = 3;
        num_comp++;
        // cout << "num_comp = " << num_comp << endl;

        }
        else {
              // Add (v,lowpt1[w],j) to ESTACK
      ADD_E(v,LOWPT1[w],j);
              DEGREE[v]++;
              DEGREE[LOWPT1[w]]++;
             }
      } else {

j++;

        // cout << "num_comp = " << num_comp << endl;
/* cout << "----- NEW SPLIT COMPONENT ----------------
cout << "(" << v << "," << LOWPT1[w] << "," << j-1 << "
```

```
 cout << "(" << v << "," << LOWPT1[w] << "," << 0 << ")
 cout << "(" << v << "," << LOWPT1[w] << "," << j << ")"
         */

         MARK[v][LOWPT1[w]] = MARK[LOWPT1[w]][v] = j;
         // cout << "Mark [" << v << "][" << LOWPT1[w] <

         C[num_comp].E[0].u = v;
 C[num_comp].E[0].v = LOWPT1[w];
         C[num_comp].E[0].z = j-1;
 C[num_comp].E[1].u = v;
 C[num_comp].E[1].v = LOWPT1[w];;
         C[num_comp].E[1].z = 0;
 C[num_comp].E[2].u = v;
 C[num_comp].E[2].v = LOWPT1[w];;
         C[num_comp].E[2].z = j;
 C[num_comp].type = BONDS;   // Multilink ..
 C[num_comp].e = 3;
 num_comp++;

         // getchar();

}

      }  // end type 1 pair loop ..

     int check3=0;
     for(g=0;g<path;g++)    // for each path ...
```

```
if((v==Paths[g].P[0].u)&&(w==Paths[g].P[0].v))
          // if (v,w) is a first edge of a path
  check3=1;

      if(check3==1){

// cout << "(" << v << "," << w << ") is a first edge" <<
        // cout << "Delete all entries down to EOS .. " <

check = 1;
while(check==1){

  if(tstack==0) check = 2;

  else if ((TSTACK[tstack-1].h == 0) && (TSTACK[tstack-1]
   && (TSTACK[tstack-1].b == 0 )){
    DELETEA(TSTACK[tstack-1].h,TSTACK[tstack-1].a,TSTACK[
    check = 2;
  }
  else  DELETEA(TSTACK[tstack-1].h,TSTACK[tstack-1].a,TST

  if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
    curr_b = TSTACK[tstack-1].b;
  }
  else curr_h = curr_a = curr_b = 0;
```

```
}

      }

      // while (h,a,b) on TSTACK has HIGHPT[v] > h ..
      while((HIGHPT[v] > curr_h) && (tstack>0)) {

DELETEA(curr_h,curr_a,curr_b);

if(tstack>0){
  curr_h = TSTACK[tstack-1].h;
  curr_a = TSTACK[tstack-1].a;
  curr_b = TSTACK[tstack-1].b;
}
else curr_h = curr_a = curr_b = 0;

      }

    }
    else {       // v - -> w

      // if (v,w) is the first and last edge of a path ..

      for(g=0;g<path;g++)
if((v==Paths[g].P[0].u)&&(w==Paths[g].P[0].v)&&(Paths[g].

  del_check = y = 0;
```

```
if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
    curr_b = TSTACK[tstack-1].b;
}
else curr_h = curr_a = curr_b = 0;

while((curr_a > w) && (tstack>0)){

  y = max(y,curr_h);
  DELETEA(curr_h,curr_a,curr_b);

  del_check = 1;

  if(tstack>0){
    curr_h = TSTACK[tstack-1].h;
    curr_a = TSTACK[tstack-1].a;
    curr_b = TSTACK[tstack-1].b;
  }
  else curr_h = curr_a = curr_b = 0;

}

if(del_check==0) {      // Add (v,w,v) to TSTACK
  ADD(v,w,v);
}
else if ( (Last_Deleted.h == h) && (Last_Deleted.a == a
                && (Last_Deleted.b == b)) {
```

```
    // Add (y,w,b) to TSTACK
    ADD(y,w,b);
  }
}

      if(w==FATHER[v]){

j = j+1;

// Add (v,w), (v,w,j), tree arc (w,v) to new component ..

        /* cout << "num_comp = " << num_comp << endl;
cout << "NEW COMPONENT ------- E ------------" << endl;
cout << "(" << v << "," << w << ")" << endl;
cout << "(" << v << "," << w << "," << j << ")" << endl;

// Mark tree arc (w,v) as virtual edge j
cout << "(" << v << "," << w << "," << j << ")" << endl;
        */

C[num_comp].E[0].u = v;
C[num_comp].E[0].v = w;
C[num_comp].E[1].u = v;
C[num_comp].E[1].v = w;
C[num_comp].E[2].u = v;
C[num_comp].E[2].v = w;
C[num_comp].type = BONDS;  // Multilink ...
C[num_comp].e = 3;
```

```
num_comp++;

        // getchar();

DEGREE[v]--;
DEGREE[w]--;


      }
      else  {
        r = MARK[v][w];
        ADD_E(v,w,r);   // Add (v,w) to ESTACK
     }

    }

  }
}


void PATHFINDER(int v){  // Find a set of paths that cove

  int i,w,j;

  NEWNUM[v] = m - ND[v] + 1;

  for(i=0;i<A_1[v].deg;i++){
```

```
    w = A_1[v].neigh[i];

    if(s==START) {    // start new path ..
      s = v;
      path++;
    }

    // add (v,w) to current path

    Paths[path-1].P[path_cnt].u = v;
    Paths[path-1].P[path_cnt].v = A_1[v].neigh[i];
    Paths[path-1].cnt++;

    if(Edge[v][w]==1){   // if v -> w

      path_cnt++;
      PATHFINDER(w);
      m--;

    }

    else if(Edge[v][w]==2) {          // if v - -> w

      if(HIGHPT[NEWNUM[w]]==0) {

HIGHPT[NEWNUM[w]] = NEWNUM[v];

      }
```

```
      // output current path

      s = START;
      path_cnt = 0;

    }
  }
}

void DFS(int v, int u){

  int i=0,w;

  n = NUMBER[v] = n + 1;

  // (comment a) ..
  LOWPT1[v] = LOWPT2[v] = NUMBER[v];
  ND[v] = 1;
  // ( end comment a ) ..

  for(i=0;i<A[v].deg;i++){    // for each w in A(v)

    if(NUMBER[A[v].neigh[i]]==0){   // w is a new vertex

      Edge[v][A[v].neigh[i]] = 1;   // (v,w) is a tree ar

      DFS(A[v].neigh[i],v);
```

```
      // (comment b) ..
      if(LOWPT1[A[v].neigh[i]] < LOWPT1[v]){
LOWPT2[v] = min(LOWPT1[v],LOWPT2[A[v].neigh[i]]);
LOWPT1[v] = LOWPT1[A[v].neigh[i]];
      }
      else if(LOWPT1[A[v].neigh[i]]==LOWPT1[v]){
LOWPT2[v] = min(LOWPT2[v],LOWPT2[A[v].neigh[i]]);
      }
      else {
LOWPT2[v] = min(LOWPT2[v],LOWPT1[A[v].neigh[i]]);
      }

      ND[v] = ND[v] + ND[A[v].neigh[i]];

      FATHER[A[v].neigh[i]] = v;

      // end (comment b ) ....

    }


    else if ( (NUMBER[A[v].neigh[i]] < NUMBER[v]) &&
      ( (A[v].neigh[i] != u) || (FLAG[v]==1) ) ){


      Edge[v][A[v].neigh[i]] = 2;   // (v,w) is a frond
```

```
      // (comment c) ..
      if(NUMBER[A[v].neigh[i]] < LOWPT1[v]){
LOWPT2[v] = LOWPT1[v];
LOWPT1[v] = NUMBER[A[v].neigh[i]];
      }
      else if(NUMBER[A[v].neigh[i]] > LOWPT1[v]){
LOWPT2[v] = min(LOWPT2[v],NUMBER[A[v].neigh[i]]);
      }
      // end (comment c ) ...

    }
    if(A[v].neigh[i]==u) FLAG[v]=1;

  }

}

void Create_Adj(void){

  //  ------- Create adjacency structure for the split co

  int j=0,k=0,l=0,h=0;

  for(j=0;j<num_comp;j++)     // for each component
    for(k=0;k<C[j].e;k++)   // for each edge
      if(C[j].E[k].z != 0)   // if its virtual

        for(l=j+1;l<num_comp;l++)  // for every other co
```

```
            for(h=0;h<C[l].e;h++)   // for each edge

            // If components i and j share a virtual edge
            // place (u,v,j) in A[i][j], A[j][i]
               if(C[l].E[h].z == C[j].E[k].z){
                   Comp_A[j][l].u = Comp_A[l][j].u = C[j].
                   Comp_A[j][l].v = Comp_A[l][j].v = C[j].
                   Comp_A[j][l].z = Comp_A[l][j].z = C[j].

                   Comp_D[j]++; Comp_D[l]++;
              }

}

void Merge_Poly(void){

  //  --------------- Merge adjacent polygons ----------
  int i=0,j=0,m=0,n=0;

  for(i=0;i<num_comp;i++)
     if(strcmp(Type[C[i].type],"POLYGONS")==0)      // fo
        for(j=0;j<num_comp;j++){                    // fo

           // which is a neighbor & polygon
           if( (strcmp(Type[C[j].type],"POLYGONS")==0)
                  && (Comp_A[i][j].u != 0) ) {

              // Add edges of j except virtual edge
```

```
// (Comp_A[i][j].u,Comp_A[i][j].v,Comp_A[

for(int m=0;m<C[j].e;m++){
   // if edge e not virtual, add to comp
   if(C[j].E[m].z != Comp_A[i][j].z){
     C[i].E[C[i].e].u = C[j].E[m].u;
     C[i].E[C[i].e].v = C[j].E[m].v;
     C[i].E[C[i].e].z = C[j].E[m].z;
     C[i].e++;        // increase edges i
     }
 }

// Delete virtual edge in comp i ...
for(m=0;m<C[i].e;m++)
   if(C[i].E[m].z == Comp_A[i][j].z){

        for(n=m;n<(C[i].e - 1);n++){
           C[i].E[n].u = C[i].E[n+1].u;
           C[i].E[n].v = C[i].E[n+1].v;
           C[i].E[n].z = C[i].E[n+1].z;
        }

        C[i].E[ C[i].e - 1 ].u = 0;
        C[i].E[ C[i].e - 1 ].v = 0;
        C[i].E[ C[i].e - 1 ].z = 0;

   }
C[i].e--;     // Decrease edges in comp i
```

```
// Update adjacency structures of neighbo
for(m=0;m<num_comp;m++)

   if(Comp_A[m][j].u != 0){

       if(m == i){

        Comp_A[m][j].u = Comp_A[m][j].v =
        Comp_A[j][m].u = Comp_A[j][m].v =
        Comp_D[i]--;

       }
       else{  // neighbors of j ...

        Comp_A[i][m].u = Comp_A[m][i]. u
        Comp_A[i][m].v = Comp_A[m][i]. v
        Comp_A[i][m].z = Comp_A[m][i]. z
        Comp_D[i]++;

        Comp_A[m][j].u = Comp_A[m][j].v =
     }
 }

// Set Comp_A[j][0...numcomp] to 0
for(m=0;m<num_comp;m++)
   Comp_A[j][m].u = Comp_A[j][m].v = Comp
```

```
            }
        }

   getchar();


}

void Acc_Adj(void){

  // Initialize variables and Structures ...
  int curr_v = 0,prev_v = 0,temp_v = 0;
  int i=0,j=0,v1=0,w1=0;

  for(i=0;i<(2*nodes)+1;i++)
    BUCKET[i].num = 0;

  // ------------ STEP 2 --------------------------- //

  for(i=1;i<nodes+1;i++)
    for(j=1;j<nodes+1;j++){

      if(Edge[i][j]==1) {

// compute theta(v,w) for tree arcs
if(LOWPT2[j] < i){
  theta[i][j] = 2 * LOWPT1[j];
}
```

```
if(LOWPT2[j] >= i){
  theta[i][j] = 2 * LOWPT1[j] + 1;
}

BUCKET[theta[i][j]].Edges[BUCKET[theta[i][j]].num].u = i;
BUCKET[theta[i][j]].Edges[BUCKET[theta[i][j]].num].v = j;

BUCKET[theta[i][j]].num++;
      }

      if(Edge[i][j]==2) {

// compute theta(v,w) for fronds

        theta[i][j] = 2 * j + 1;

BUCKET[theta[i][j]].Edges[BUCKET[theta[i][j]].num].u = i;
BUCKET[theta[i][j]].Edges[BUCKET[theta[i][j]].num].v = j;

BUCKET[theta[i][j]].num++;


      }

    }

   for(j=0;j<nodes+1;j++)
```

```
      A_1[j].deg = 0;

    for(i=0;i<(2*nodes)+1;i++){
      prev_v = 0;

     for(j=0;j<BUCKET[i].num;j++){

       v1 = BUCKET[i].Edges[j].u;
       w1 = BUCKET[i].Edges[j].v;

       curr_v = v1;

       // Add w1 to end of A1[v]
       A_1[v1].neigh[A_1[v1].deg] = w1;

       if(curr_v == prev_v){

if((A_1[curr_v].neigh[A_1[curr_v].deg] > curr_v)
   &&(A_1[curr_v].neigh[A_1[curr_v].deg-1] < curr_v)){

  temp_v =  A_1[curr_v].neigh[A_1[curr_v].deg-1];

  A_1[curr_v].neigh[A_1[curr_v].deg-1] =
    A_1[curr_v].neigh[A_1[curr_v].deg];

  A_1[curr_v].neigh[A_1[curr_v].deg] = temp_v;

}
```

```
      }

      A_1[v1].deg++;
      prev_v = curr_v;

    }
  }
}

// Main Function ...

int main(int argc, char* argv[]){

  int i=0, curr_v, prev_v, temp_v;   // variables used
  int m=0,n=0;

  // Sample input, |V|=6, |E| = 9

    A[1].deg = 2;
    A[1].neigh[0] = 2;
    A[1].neigh[1] = 3;

    A[2].deg = 3;
    A[2].neigh[0] = 1;
    A[2].neigh[1] = 3;
    A[2].neigh[2] = 4;

    A[3].deg = 4;
```

```
  A[3].neigh[0] = 1;
  A[3].neigh[1] = 2;
  A[3].neigh[2] = 4;
  A[3].neigh[3] = 5;

  A[4].deg = 4;
  A[4].neigh[0] = 2;
  A[4].neigh[1] = 3;
  A[4].neigh[2] = 5;
  A[4].neigh[3] = 6;

  A[5].deg = 3;
  A[5].neigh[0] = 3;
  A[5].neigh[1] = 4;
  A[5].neigh[2] = 6;

  A[6].deg = 2;
  A[6].neigh[0] = 4;
  A[6].neigh[1] = 5;

  nodes=6;m=6;

// Initialize DFS(), PATHFIND(), and PATHSEARCH() Struc

for(i=1;i<nodes+1;i++){
  NUMBER[i] = FLAG[i] = Paths[i].cnt = 0;
  LOWPT1[i] = LOWPT2[i] = ND[i] = FATHER[i] = 1;
  for(j=1;j<nodes+1;j++)
```

```
    Edge[i][j]=0;
 }

// Initialize Triconnected Component Structures ...

for(i=0;i<15;i++){
   Comp_D[i]=0;
   for(j=0;j<15;j++)
      Comp_A[i][j].u = Comp_A[i][j].v = Comp_A[i][j].z
   for(j=0;j<30;j++)
      Comp_A[i][j].u = Comp_A[i][j].v = Comp_A[i][j].z
 }
for(i=0;i<10;i++)
  SepPair[i].u = SepPair[i].v = 0;

// Initialize Graph Drawing Structures ...

for(i=0;i<20;i++){
   C[i].face_v = C[i].inner_v = C[i].d_bond = 0;
   Final.face_v = Final.inner_v = 0;
   for(j=0;j<30;j++){
     C[i].E[j].u = C[i].E[j].v = C[i].E[j].z = 0;
     C[i].F[j] = C[i].I[j] = C[i].V[j] = 0;
     Final.E[j].u = Final.E[j].v = Final.E[j].z = 0;
     Final.F[j] = Final.I[j] = 0;
     T_number[j] = 0;
   }
   for(j=0;j<20;j++)
```

```
      MARK[i][j] = 0;
  }

  // ---- BEGIN ----------------------------------------

  // STEP 1 - Depth First Search / Construct Palm Tree
  //        - Calculate LOW1, LOW2, ND and FATHER values

  n = 0;
  int start=1,temp;
  DFS(start,1);         // DFS() - Generate Palm tree

  // Print out tree arcs and fronds of P
  // cout << " ----------- RESULTS - Procedure 1 --------

  cout << "Tree arcs .. ";
  for(i=1;i<nodes+1;i++)
    for(j=1;j<nodes+1;j++)
      if(Edge[i][j]==1)
cout << "(" << i << "," << j << ") ";

  cout << endl << "Fronds .. ";
  for(i=1;i<nodes+1;i++)
    for(j=1;j<nodes+1;j++)
      if(Edge[i][j]==2)
cout << "(" << i << "," << j << ") ";
  cout << endl;
```

```
// STEP 2 - Construct acceptable adjacency structure ..
Acc_Adj();

// STEP 3 - Peform a depth-first search of G using new
// Procedure 5 ....

s = START;
path_cnt=0;

for(i=1;i<nodes+1;i++){
  NEWNUM[i] = 0;
  HIGHPT[i] = 0;
  A1[i] = 0;
}

// Generate a set of disjoint paths that cover G
// Start at vertex 1 ...
PATHFINDER(1);

for(i=1;i<nodes+1;i++){        // Set the first descend
  A1[i] = A_1[i].neigh[0];     // Set the degree of ver
  DEGREE[i] = A[i].deg;

}

// Print out results of PATHFINDING() Procedure ...
cout << "------------- Procedure 5 Results -----------
cout << "------- # of Paths = " << path << " -------- "
```

```
cout << "================================" << endl;
for(i=0;i<path;i++){
  for(j=0;j<Paths[i].cnt;j++)
    cout << "(" << Paths[i].P[j].u << "," << Paths[i].P
  cout << endl;
}

getchar();

// Procedure 6 - Determine Split Components ...
// Initialize Stacks ...
for(i=0;i<50;i++){
  ESTACK[i].u = ESTACK[i].v = ESTACK[i].z = 0;
  TSTACK[i].h = TSTACK[i].a = TSTACK[i].b = 0;
}

j=0;
FLAG1 = FALSE;

// ------ PATHSEARCH -- Generate Split Components -----
PATHSEARCH(1);

// cout << "------ FINAL COMPONENT ----------" << endl;
for(i=0;i<estack;i++){
  // cout << "(" << ESTACK[i].u << "," << ESTACK[i].v <
  C[num_comp].E[i].u = ESTACK[i].u;
  C[num_comp].E[i].v = ESTACK[i].v;
  C[num_comp].E[i].z = ESTACK[i].z;
```

```
}
// cout << endl;   // # of virtual edges = j;
C[num_comp].e = estack;

// Check the type of this new split component ..
// Check degree of first vertex u found in the first ed

v_cnt = 0; v_one = 0; c = 0;
for(c=0;c<C[num_comp].e;c++)  {
  if(c==0){ v_one = C[num_comp].E[0].u; v_cnt++; }
      else { if( (C[num_comp].E[c].u == v_one) || (C[nu
      v_cnt++;
      } }

if(v_cnt > 2) C[num_comp].type = TRI;    // Tri.
else C[num_comp].type = POLYGONS;    // Triconnected Co


num_comp++;
cout << "There are " << num_comp << " split components
for(i=0;i<num_comp;i++)
  cout << "C[" << i << "] has " << C[i].e << " edges, t
cout << endl;

getchar();

// -------------------- Print out the split components
for(j=0;j<num_comp;j++){
```

```
   cout << "[" << j << "] ";
   for(i=0;i<C[j].e;i++)
     cout << " (" << C[j].E[i].u << "," << C[j].E[i].v
   cout << Type[C[j].type] << endl;
}

getchar();

//  ..... ..... .... ..... ....... .... Print out the
cout << "Separation Pairs .... {";
for(i=0;i<seppair;i++)
   cout << "(" << SepPair[i].u << "," << SepPair[i].v <
cout << "}" << endl;

// ------------------------------------------------
// Rearrange split components to create triconnected co
// Set Up Component Adjacency Structure .. Create virtu
// ----- ----- ---- ---------------------- Create 3-blo

// --- Create Adjacency Structure for Split Components
Create_Adj();

//  --------------- Merge adjacent polygons ----------
Merge_Poly();

// getchar();

// --- ------------ ------ Print out triconnected compo
```

```
int new_count=0;
for(j=0;j<num_comp;j++)
   if(Comp_D[j] > 0){
     new_count++;
     cout << "Comp [" << j << "].D = " << Comp_D[j] <<
     for(i=0;i<C[j].e;i++)
       cout << "(" << C[j].E[i].u << "," << C[j].E[i].v
     cout << Type[C[j].type] << endl;
  }

cout << "There are now " << new_count << " triconnected
getchar();

// Draw each Triconnected Graph G and Polygons and Mult

// m_edge = nodes + 1;

for(j=0;j<num_comp;j++)
   if(Comp_D[j] > 0)
      Draw_T(j);    // Draw each triconnected comp

// Merge planar representations ..
Merge_3_Block_Tree(new_count);

getchar();

// Dual Eulerian Alg. Setup ..........................
```

```
int a=0,b=0,next=0;
V=nodes; DUAL=0;

// Initialize arrays ...
for(i=0;i<30;i++){
  SPECIAL[i] = 0;
  Vrt[i].u = Vrt[i].v = Vrt[i].id = Vrt[i].part = 0;
  for(j=0;j<30;j++)
    W[i][j] = 0; }
for(i=0;i<30;i++){
  Tr[i].x = Tr[i].y = 0;
  C1[i].x = C1[i].y = 0;
  C2[i].x = C2[i].y = 0;
  Temp[i].x = Temp[i].y = 0; }

// Copy Final[x,y] into C1[x,y] ....
for(j=1;j<nodes+1;j++){
   C1[j].x = Final.Pos[j].x;
   C1[j].y = Final.Pos[j].y;
}

int k=0;
// Set up adjacency matrix from adjacency lists ...
for(i=1;i<nodes+1;i++)         // for each vertex i ..
  for(k=i+1;k<nodes+1;k++)       // for each vertex k .
    for(j=0;j<A[i].deg;j++)  // for each neighbor of i
           if(A[i].neigh[j] == k) { W[i][k]++; W[k][i]+
```

```
/* Print out adjacency matrix ...
for(i=1;i<nodes+1;i++){
  for(k=1;k<nodes+1;k++)
      cout << W[i][k] << " ";
  cout << endl;
} */

// ---------- Begin (Multiple Edge Section) ----------
// Creates 2 virtual edges for each multiedge, and the
// algorithm keeps track of the edges that are virtual
// For each multiple edge ... create virtual edges part

int cnt=0; int m_edge=0;
for(i=1;i<nodes+1;i++)
  for(k=i+1;k<nodes+1;k++)              // note .. i < k
      if(W[i][k] > 1){
         int part=0;
         for(j=0;j<(W[i][k]-1);j++) {  // for each mult

            Vrt[m_edge].u = i;        // part 1
            Vrt[m_edge].v = k;
            Vrt[m_edge].id = j;
            Vrt[m_edge].part = part;

            m_edge++;
            part++;

            Vrt[m_edge].u = i;       // part 2
```

```
            Vrt[m_edge].v = k;
            Vrt[m_edge].id = j;
            Vrt[m_edge].part = part;

            m_edge++;
          }
        }


cnt=1;
for(i=0;i<m_edge;i++){          // for every virtual edg

   // Print out first part of each pair of virtual egde
   cout << "(" << Vrt[i].u << "," << Vrt[i].v << ","
        << Vrt[i].id << "," << Vrt[i].part << ") ";

   // Get coordiantes of vertices incident to multiple
   int x1 = C1[Vrt[i].u].x; int y1 = C1[Vrt[i].u].y;
   int x2 = C1[Vrt[i].v].x; int y2 = C1[Vrt[i].v].y;

   // Create virtual vertices corr. to virtual edges ..
   // New vertex  is (V+cnt) ..

   if(x1==x2){          // vertical line
       cout << " vert ";
       C1[V+cnt].x = x1 + .5;
       C1[V+cnt].y = (y1+y2)/2;
    }
```

```
else if(y1==y2){       // horizontal line
    cout << " horiz ";
    C1[V+cnt].x = (x1+x2)/2;
    C1[V+cnt].y = y1 + 0.5;
 }
else {              // slope > 0 or slope < 0
    cout << " s>0 or s<0 ";
    C1[V+cnt].x = x1 + .75*(x2-x1);
    C1[V+cnt].y = min(x1,x2) + (.75)*(max(x1,x2)-min
}

// Print out new vertex information ...
cout << V+cnt << " (x,y) = " << "(" <<  C1[V+cnt].x
       << "," << C1[V+cnt].y << ")" << endl;

// Update adjacency structures of (V+cnt), (Vrt[i].u
A[V+cnt].deg = 2;
A[V+cnt].neigh[0] = Vrt[i].u; A[V+cnt].neigh[1] = Vr

A[Vrt[i].u].neigh[ A[Vrt[i].u].deg ] = V+cnt;
A[Vrt[i].u].deg++;

A[Vrt[i].v].neigh[ A[Vrt[i].v].deg ] = V+cnt;
A[Vrt[i].v].deg++;

// Mark New virtual vertices as SPECIAL ..
SPECIAL[nodes+cnt] = 1;
```

```
   i++;       // Skip second part of virtual edge ...
   cnt++;     // Increase next virtual vertex number by
}

// ----------------- Update |V| and |E| --------------
V = V + (cnt-1);
E = E + (m_edge/2);
cout << " ----------- |V|= " << V << " |E|= " << E << "

// ---------------------- End (Multiple Edge Section)

// Copy C1[] to C2[] (temp lists)
for(int i=1;i<(V+1);i++){
  C2[i].x = C1[i].x;
  C2[i].y = C1[i].y;
}

// Dual Eulerian Algorithm .....
// Finds the maximum size Petrie path in G. This versio
// to build the path from the starting edge, going the
// starting with the same initial turn.

int flag=0;    // Checks when to stop euler petrie path
int TRY=0;     // From (a,b), go left ... then if necce

turn = LEFT;   // First turn left first ...

while( (DUAL==0) && (TRY<2)) {
```

```
 getchar();

// Initialize edge stacks ...
for(i=0;i<30;i++)
  STACK[i].u = STACK[i].v = 0;

RESET();    // Reset coordinates of each vertex ..

// Begin with first two vertices ... (v_1,v_2) , v_2

a = 1; b = A[1].neigh[0];
Strt.u = a; Strt.v = b;
STACK[0].u = a; STACK[0].v = b;  stack_cnt = 1;

while(flag==0){      //  Euler-Petrie algorithm ....

  cout << "(" << a << "," << b << ") ";

  if(turn==LEFT) cout << " L " << endl;
  else cout << " R " << endl;

  RESET();

  TURN(a,b,turn);   // turn (turn) at edge (a,b)

  if(turn==LEFT) next=DIR.left;
  else next=DIR.right;
```

```
      // Check for multiple edge on STACK ..
      // cout << "Checking (" << b << "," << next << ")"

      if( STACK_CHECK(b,next) == 1)  flag = 1;

      else {       // Add new edge to STACK ...

STACK[stack_cnt].u = b;
STACK[stack_cnt].v = next;
stack_cnt++;

a = b; b = next;

        if(SPECIAL[next]==0){          // if current head
  if(turn==LEFT) turn=RIGHT;   // switch direction ..
          else turn=LEFT;
        }

      }

    }


    TRY++;
    flag = 0;       // Re-enter previous loop ..
    turn = RIGHT;   // If neccessary turn right first ...
```

```
  } // end main while()

  getchar();


  return(0);

}  // end of main program ..

 /clearpage

// Biconnectivity Algorithm ..
// Input: Undirected graph without multiple edges ..
// Output: Biconnected Components ...
// Implementation of created by Robert Tarjan in [10]
// Store biconnected components in B[].
// # of biconnected components is bicmps.

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream.h>
#include <algorithm>
#include <conio.h>

#define START 1000
#define EOS 100
#define TRUE 1
```

```
#define FALSE 0

int nodes = 0, m = 0;   // # of vertices in G

typedef struct {      // Adjacency lists for each vertex

  int neigh[50];     // List of vertices adjacent to v
  int deg;           // Degree of vertex v

} Adj_List;

typedef struct {       // Structure

  int u; int v; int z;

} EDGE;

typedef struct {       // Stores each biconnected compone

  EDGE edges[20];      // Stores edges of component B
  int e;               // The number of edges in componen

} Bi_Comp;

typedef struct {
   int u; int v; int d;
} Bond;
```

```
EDGE AllEdges[50];   // Store edges ..
int alledge = 0;

Bond Bonds[20];      // Store Bonds
int bonds = 0;

Bi_Comp B[20];       // Biconnected Components ...
int bicmps = 0;      // # of biconnected components ..

EDGE STACK[50];      // Edge stack ...
Adj_List A[50];      // Adjacency lists ...

int st_cnt=0;        // Edge count for STACK
int W[50][50];       // Weight Matrix ...
int NUMBER[50];      // Vertex Numbering ...
int LOWPT1[50];      // LOWPT1 Info ...
int b_cnt=0;         // Counter for Biconnect() numbering

void BICONNECT(int v, int u);

void BICONNECT(int v, int u){

  int w,u1,u2,u3,cnt=0;

  NUMBER[v] = b_cnt = b_cnt + 1;
  LOWPT1[v] = NUMBER[v];

  for(int i=0;i<A[v].deg;i++){
```

```
    w = A[v].neigh[i];

    if(NUMBER[w]==0){   // If w is not numbered ..

      // Add (v,w) to STACK ..
      STACK[st_cnt].u = v;
      STACK[st_cnt].v = w;
      st_cnt++;

      BICONNECT(w,v);

      LOWPT1[v] = min(LOWPT1[v],LOWPT1[w]);

      if(LOWPT1[w] >= NUMBER[v]){

// Start new biconnected component ...

cnt=0;

u1 = STACK[st_cnt-1].u;
u2 = STACK[st_cnt-1].v;

while(NUMBER[u1] >= NUMBER[w]){

  // Send (u1,u2) to new component ..
  cout << "(" << u1 << "," << u2 << ")";
```

```
  B[bicmps].edges[cnt].u = u1;
  B[bicmps].edges[cnt].v = u2;
  cnt++;

  // Delete (u1,u2) from STACK ..
  STACK[st_cnt-1].u = 0;
  STACK[st_cnt-1].v = 0;
  st_cnt--;

  u1 = STACK[st_cnt-1].u;
  u2 = STACK[st_cnt-1].v;

}

cout << "(" << v << "," << w << ")." << endl;

B[bicmps].edges[cnt].u = v;
B[bicmps].edges[cnt].v = w;
cnt++;

B[bicmps].e = cnt;  // Set number of edges in new compone

bicmps++;  // Increase # of components by 1 ...

STACK[st_cnt-1].u = 0;
STACK[st_cnt-1].v = 0;
st_cnt--;
```

```
      }
    }
    else if(  (NUMBER[w] < NUMBER[v]) && (w != u) ){


      // Add (v,w) to STACK
       STACK[st_cnt].u = v;
       STACK[st_cnt].v = w;

       st_cnt++;

      LOWPT1[v] = min(LOWPT1[v],NUMBER[w]);

    }
  }
}

int main(){

  int i=0,j=0,k=0;

   // Initialize adjacency structures W & A to 0
   // W is adjacency matrix. A[] is set of adjacency list

   for(i=0;i<50;i++){
      A[i].deg = 0;
      AllEdges[i].u = AllEdges[i].v = AllEdges[i].z = 0;
      for(j=0;j<50;j++){
```

```
A[i].neigh[j] = 0;
W[i][j] = 0;
      }
    }

    // Initialize arrays ..
    for(i=0;i<50;i++)
      NUMBER[i] = LOWPT1[i] = 0;

    // Initialize edge stack ..
    for(i=0;i<50;i++)
      STACK[i].u = STACK[i].v = 0;

    // Initialize Biconnected Components ...
    for(i=0;i<20;i++){
      B[i].e = 0;
      Bonds[i].u = Bonds[i].v = Bonds[i].d = 0;
      for(j=0;j<20;j++)
B[i].edges[j].u = B[i].edges[j].v = B[i].edges[j].z = 0;
    }

     bicmps = b_cnt = st_cnt = bonds = 0;   // Reset coun

     // Input Adjacency Structure (Weight Matrix W) ..
     // Example

      W[1][2] = W[1][5] = W[1][6] = W[1][7] = 1;
      W[2][1] = W[2][3] = W[2][4] = W[2][5] = 1;
```

```
      W[3][2] = W[3][4] = 1;
      W[4][2] = W[4][3] = 1;
      W[5][1] = W[5][2] = W[5][6] = 1;
      W[6][1] = W[6][5] = 1;
      W[7][1] = W[7][8] = W[7][9] = 1;
      W[8][7] = W[8][9] = 1;
      W[9][7] = W[9][8] = 1;

      nodes = m = 9;

      // Convert W[][] to A[] & count |E| -------------

      int deg=0;alledge=0;

      for(i=1;i<nodes+1;i++){
deg = 0;
for(j=1;j<nodes+1;j++)
  if(W[i][j] > 0){         // if (i,j) is an edge ...

    A[i].neigh[deg] = j;    // Store j in A[i],neigh[]
    A[i].deg++;
    deg++;

            //  ......... Count and Stack ........
            if(j>i)
              for(k=0;k<W[i][j];k++) {
                AllEdges[alledge].u=i;
                AllEdges[alledge].v=j;
```

```
                alledge++;
                }
                if(W[i][j]>1){
                   Bonds[bonds].u = i;
                   Bonds[bonds].v = j;
                   Bonds[bonds].d = W[i][j];
                   bonds++;
                }
             }
        }

        cout << "|E|= " << alledge << endl;
        getch();

      // Print out A[] --------------------------
      for(i=1;i<nodes+1;i++){
 cout << i << " : { ";
 for(j=0;j<A[i].deg;j++) cout << A[i].neigh[j] << " ";
 cout << " }" << endl;
      }

      //  ----- Begin -- BICONNECT() -----------------

      b_cnt=0;
      for(i=1;i<nodes+1;i++)
if(NUMBER[i]==0)
  BICONNECT(i,0);
```

```
      // Print out Biconnected Components ...

      cout << "There are " << bicmps << " biconnected com
      for(i=0;i<bicmps;i++){
cout << "Component[" << i << "] = { ";
for(j=0;j<B[i].e;j++)
  cout << "(" << B[i].edges[j].u << "," << B[i].edges[j].
cout << "}" << endl;
      }

      cout << endl;
      getch();

     return 0;

}

/clearpage

// Structures ....
// File is structures.h

enum Comp_Type { BONDS,POLYGONS,TRI };
char *Type[] = {"BONDS","POLYGONS","TRI"};

typedef struct {
  int neigh[15];
  int deg;
```

```
} Adj_List;

typedef struct {
  int u; int v;
} EDGE;

typedef struct {
  int u; int v; int z;
} EDGES;

typedef struct {
  EDGE P[14];
  int cnt;
} PATH;

typedef struct {
  EDGE Edges[30];
  int num;
} Bucket;

typedef struct {
  int h; int a; int b;
} Triple;

typedef struct {
  double x;  double y;
} Pair;
```

```
typedef struct {
  int e,v;          // e = |E|, v = |V|
  EDGES E[30];      // Edges is |E|
  Comp_Type type;   // Type of Component - (Polygon, Mult
  int face_v;       // # of vertices on outer face ..
  int inner_v;      // # of vertices in inner face ..
  int F[30];        // outer face vertices {0 .. face_v}
  int I[30];        // Inner vertices ..
  int V[30];        // Identify Virtual vertices for mult
  int d_bond;       // Check for drawing bond ..
  Pair Pos[15];     // Vertex positions ..

  // separation pairs
  // neighbor ...

} Component;

typedef struct {

  int left;
  int right;

} Direction;

typedef struct {

   int u;
   int v;
```

```
    int id;
    int part;

} VIRT;
```

References

Bibliography

[1] B. Carlson, C. Chen, C. Meliksetian, Dual Eulerian properties of plane multigraphs, SIAM J. Discrete Math. 8(1) (1995) 33-50.

[2] C. Droms, B. Servatius, H. Servatius, The structure of locally finite 2-connected graphs, Electron. J. Combin. R17 (1995).

[3] F. Harary, Graph Theory, Addison-Wesley, (1972).

[4] J.E. Hopcraft, R.E. Tarjan, Dividing a graph into triconnected components, SIAM J. Comput. 3(2) (1973) 135-158.

[5] J. Hopcroft, R. Tarjan, Efficient Planarity Testing, J. Assoc. Comput. Mach., 21(4) (1974) 549-568.

[6] B. Plestenjak, An Algorithm for Drawing Planar Graphs, Software-Prac. and Exper. 29(11) (1999) 973-984.

[7] K. Rosen, Handbook of Discrete and Combinatorial Mathematics, CRC Press, New York, (2000).

[8] B. Servatius, H. Servatius, A polynomial time algorithm for determining the zero euler-petrie genus of an eulerian graph, Disc. Math. 244 (2002) 445 - 454.

[9] B. Servatius, H. Servatius, Dual-Eulerian Graphs, SIAM J. Discrete Math., to appear.

[10] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 2(1) (1972) 146-160.

[11] R. Wilson, Introduction to Graph Theory, John Wiley & Sons, New York, (1985).